

«Это новая книга по CSS для нового поколения CSS-разработчиков. Я лично знаком с несколькими по-настоящему гениальными разработчиками, но из них только Леа Веру способна донести до читателя свежие идеи CSS в полном объеме».

Джеффри Зельдман

СЕКРЕТЫ CSS

ИДЕАЛЬНЫЕ РЕШЕНИЯ ЕЖЕДНЕВНЫХ ЗАДАЧ

ЛЕА ВЕРУ





CSS SECRETS

BETTER SOLUTIONS TO EVERYDAY WEB DESIGN PROBLEMS

LEA VEROU

Beijing • Boston • Farnham • Sebastopol • Tokyo



ЛЕА ВЕРУ

СЕКРЕТЫ CSS

ИДЕАЛЬНЫЕ РЕШЕНИЯ ЕЖЕДНЕВНЫХ ЗАДАЧ

O'REILLY®



Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону • Самара • Минск

2016

ББК 32.988.02-018
УДК 004.738.8
В35

Веру Л.

В35 Секреты CSS. Идеальные решения ежедневных задач. — СПб.: Питер, 2016. — 336 с.: ил. — (Серия «Бестселлеры O'Reilly»).

ISBN 978-5-496-02082-4

Гибкий легкий код, соответствующий стандартам — его можно получить, если подойти к проблеме аналитически. Леа Веру познакомит вас с недокументированными приемами, позволяющими найти изящные решения для самого широкого круга задач веб-дизайна. В основу книги легли доклады автора на шестидесяти международных конференциях веб-разработчиков, так что она затрагивает самые актуальные темы — от взаимодействия с пользователем до типографики и визуальных эффектов.

Множество книг, доступных на сегодняшнем рынке, документируют возможности CSS от А до Я. Хорошо это или плохо, но «Секреты CSS» — не одна из них. Ее назначение — заполнить пробелы в знаниях, оставшиеся после того, как вы уже ознакомились со справочными материалами, открыть ваш разум новым способам применения функциональности, которая вам уже известна, а также познакомить вас с полезными возможностями CSS, которые не так модны и популярны, но заслуживают не меньшей любви. Главная задача этой книги — научить вас решать проблемы с помощью CSS.

12+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.988.02-018
УДК 004.738.8

ISBN 978-1449372637 англ.
ISBN 978-5-496-02082-4

ISBN 978-5-496-02082-4

Оглавление

От издательства

Все иллюстрации со значком 🎨 можно посмотреть в цветном варианте по ссылке http://goo.gl/jjlYmZ.

В память о моей маме и лучшем друге
Марии Веру (1952–2013),
слишком рано покинувшей этот мир.

Предисловие

О, старые добрые времена! В прошлом столетии у нас было всегда два браузера с поддержкой CSS, и поддерживали они весьма ограниченное подмножество функций из весьма ограниченной спецификации, поэтому полную карту того, что работало, а что нет, можно было с легкостью держать в голове. Эта карта также включала проблемы каждой реализации, поскольку ошибок и оплошностей — порой нелепых до комичности — хватало с лихвой. Да что там говорить, некоторые ошибки были настолько фундаментальными, что делали способы обработки макета в разных браузерах совершенно несовместимыми, заставляя нас придумывать целые арсеналы трюков, использующих дыры в синтаксическом анализаторе, только для того, чтобы обходить эти различия!

Погодите-ка. Старые добрые времена были просто *ужасными*. Как хорошо, что они остались в прошлом!

За последние несколько лет дела в сфере CSS значительно улучшились. Браузеры теперь по большей части совместимы друг с другом, а если и возникает несовместимость, то практически всегда по той причине, что один браузер не поддерживает возможность, которая уже есть в другом, а не как раньше — когда оба пытались поддерживать одну и ту же вещь, но по-разному, и чаще всего одинаково плохо. Спецификации значительно расширились и усовершенствовались и, среди прочего, включают возможности, воссоздающие хитроумные трюки из прошлого, но значительно более простыми и компактными способами. CSS предлагает гораздо больше возможностей и гораздо больше мощи, чем когда-либо до этого, — но, как все мы знаем, с большим могуществом приходят большие сложности. И дело даже не в том, что сложность повышается намеренно: когда вы

объединяете достаточно большое число работающих частей, неважно, насколько проста каждая из них, — их сочетание может породить весьма интересные вещи (эта тема хорошо раскрыта в фильме «Лего»).

Но именно эта непреднамеренная сложность дарует CSS способность удивлять нас новыми, внезапно появляющимися возможностями, которых мы не то что не ожидали, но даже не планировали. Скрещивая различные свойства и неожиданным образом используя значения, мы откроем еще множество секретов. Вы можете вырезать фигурные углы с помощью градиентов, анимировать элементы, увеличивать области, реагирующие на щелчки мыши, даже создавать секторные диаграммы... и это далеко не полный список. CSS предлагает возможности, о которых во времена, когда я был еще юным пареньком, мы могли только мечтать, возможности за пределами нашего воображения. Мы получили функциональность, которая, как мне казалось, в принципе не может быть выражена в компактной, удобной для человеческого восприятия манере — например, анимацию. CSS превратилась в настолько продвинутую технологию, что, я уверен, нас ждет еще масса удивительных открытий! Возможно, какие-то секреты удастся разоблачить именно вам.

Но пока этот день не наступил, у вас есть возможность пользоваться множеством интереснейших техник, которые уже были обнаружены и описаны, — и мало кто для этого сделал больше, чем Леа Веру. Публикации в ее блоге, ее вклад в разработку продуктов с открытым кодом, ее динамичные интерактивные выступления по всему миру — все это позволило Леа накопить огромный объем знаний о CSS. Эта книга — великолепная выжимка из ее необъятного опыта. В своих руках вы сейчас держите ключ к одним из самых интересных, удивительных и полезных техник, наработанных специалистами CSS вплоть до сегодняшнего дня, руководство, составленное ярчайшими профессионалами своего дела. То, что Леа приготовила для вас на этих страницах, обогатит вас, доставит вам наслаждение и — да — даже поразит вас!

Идите, учитесь и срывайте покровы тайны с этих открытий.

Эрик А. Мейер

Введение

За последние несколько лет язык CSS претерпел огромные преобразования, схожие с революцией JavaScript в 2004 году. Из простейшего языка стилизации с ограниченными возможностями он превратился в сложную технологию, определяемую более чем 80 спецификациями W3C (включая черновики) с собственной экосистемой разработки, собственными конференциями, собственной инфраструктурой и инструментарием. Технология CSS выросла до такого размера, что одному человеку практически невозможно целиком удержать ее в своей голове. Даже в рабочей группе CSS W3C, которая создает определение языка, никто не может назвать себя экспертом по всем возможным аспектам CSS, и лишь немногие подбираются к этому званию. Большинство членов рабочей группы фокусируются на определенных спецификациях CSS и могут практически ничего не знать об остальных.

Примерно до 2009 года квалификация специалиста по CSS определялась не его знанием языка. Это считалось данностью для любого серьезного проекта, включающего CSS. Вместо этого мерилом мастерства было количество ошибок и обходных путей, которые человек держал в памяти. Перенесемся в 2015 год: сегодня браузеры разрабатываются с учетом стандартов, и корявые трюки, завязанные на особенности конкретных браузеров, порицаются общественностью. Разумеется, периодически еще встречаются ситуации, в которых несовместимостей не избежать, но — особенно с учетом того, что большинство браузеров теперь обновляются автоматически, — темп перемен настолько высок, что попытка задокументировать их в книге была бы пустой тратой времени и места на страницах.

Трудности современного CSS не связаны с поиском обходных путей вокруг трудноуловимых ошибок в браузерах. Сложность в том, чтобы творчески применять доступные нам возможности CSS, создавая **гибкие**, **легкие решения**, **удобные в сопровождении и соответствующие принципам DRY** и, насколько возможно, **совместимые с существующими стандартами**. Именно об этом я рассказываю в книге.

Множество книг, доступных на сегодняшнем рынке, документируют определенные возможности CSS от А до Я. Хорошо это или плохо, но «Секреты CSS» — не одна из них. Ее назначение — заполнить пробелы в знаниях, оставшиеся после

того, как вы уже ознакомились со справочными материалами, открыть ваш разум новым способам применения функциональности, с которой вы уже знакомы, а также познакомить вас с полезными возможностями CSS, которые не так модны и популярны, но заслуживают не меньшей любви. Однако прежде всего главная задача этой книги — научить вас **решать проблемы с помощью CSS**.

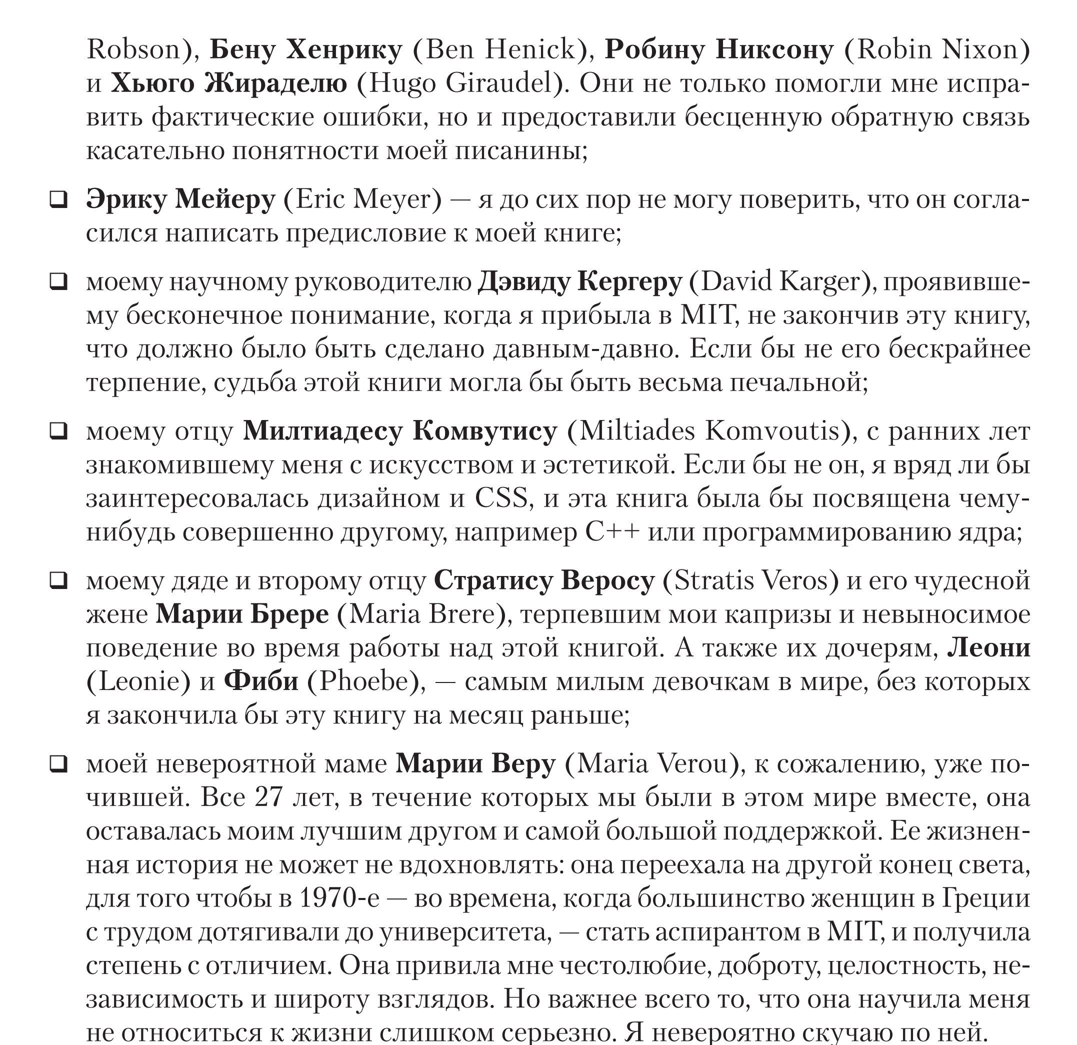
Также «Секреты CSS» не относится к классу сборников советов. Ни один из содержащихся здесь «секретов» не является завершенным рецептом, требующим строгого следования каждому шагу, для того чтобы достичь определенного эффекта. Я постаралась детально описать процесс мышления, лежащий в основе реализации каждой техники, так как я верю, что **понимание процесса поиска решения намного ценнее, чем само решение**. Даже если вы не думаете, что та или иная техника пригодится вам в вашей конкретной работе, понимание пути, по которому мы пришли к решению, может оказаться полезным при попытке справиться даже с совершенно непохожими проблемами. Короче говоря, **из этой книги вы вытащите немало пресловутых рыбок, и все же моя основная цель здесь — дать вам в руки удочку, научив, как их ловить**.

Аббревиатура **DRY** расшифровывается как *Don't Repeat Yourself* — «не повторяйтесь». Это популярная в программистском сообществе мантра, продвигающая один из аспектов удобного в сопровождении кода: возможность менять значения параметров с помощью как можно меньшего числа правок, в идеальном случае ограничиваясь одной. Акцент на соответствии CSS-кода принципам DRY — повторяющаяся тема этой книги. Противоположность DRY — принцип **WET**, что означает *We Enjoy Typing* («нам нравится печатать») или *Write Everything Twice* («пишите все дважды»).

Благодарности

Эта книга никогда бы не была написана, если бы не помощь и поддержка множества потрясающих людей, которым я бесконечно благодарна. Огромное сердечное спасибо:

- всем, кто на протяжении многих лет поддерживал меня в работе, иначе я в принципе никогда не смогла бы написать книгу. Читателям **моего блога** (http://lea.verou.me), ленты в **Twitter** (http://twitter.com/leaverou) и публикаций в любых других источниках и особенно — **тебе**, дорогой читатель моей первой книги! Всем, кто использует мои **разработки с открытым кодом** (http://github.com/leaverou), и еще больше тем, кто внес свой вклад в их создание;
- всем организаторам конференций, которые приглашали меня для проведения лекций и семинаров, особенно **Дамиану Вилгосику** (Damian Wielgosik) и **Павлу Черски** (Paweł Czerski), первыми поверившим в меня и пригласившим на инаугурационную конференцию Front-Trends в 2010 году. А также **Василису Вассалосу** (Vasilis Vassalos), который в 2010 году доверил мне подготовку курса по веб-разработке для Афинского университета экономики и бизнеса. Этот опыт оказался для меня бесценным уроком преподавания (а техническая книга — это, по сути, преподавание);
- всем членам **рабочей группы CSS**, проголосовавшим за приглашение меня в качестве внешнего эксперта, — это событие полностью изменило мои взгляды на веб-технологии в целом и CSS в частности;
- моим редакторам, **Мэри Треселер** (Mary Treseler) и **Мэг Фолей** (Meg Foley), передавшим контроль над всем процессом в мои руки и проявлявшим невероятное терпение, когда я пропускала установленные сроки (что случалось куда чаще, чем мне хотелось бы признавать);
- моему редактору по производству **Каре Ибрагим** (Kara Ebrahim), которая потратила огромное количество времени, исправляя ошибки макета и вручную убирая проблемы визуализации CSS, компенсируя ограничения программы рендеринга PDF, которая использовалась для этой книги;
- моим научным редакторам: **Элике Этемад** (Elika Etemad), **Табу Аткинсу** (Tab Atkins), **Райану Седдону** (Ryan Seddon), **Элизабет Робсон** (Elisabeth



Robson), **Бену Хенрику** (Ben Henick), **Робину Никсону** (Robin Nixon) и **Хьюго Жираделю** (Hugo Giraudel). Они не только помогли мне исправить фактические ошибки, но и предоставили бесценную обратную связь касательно понятности моей писанины;

- **Эрику Мейеру** (Eric Meyer) — я до сих пор не могу поверить, что он согласился написать предисловие к моей книге;
- моему научному руководителю **Дэвиду Кергеру** (David Karger), проявившему бесконечное понимание, когда я прибыла в MIT, не закончив эту книгу, что должно было быть сделано давным-давно. Если бы не его бескрайнее терпение, судьба этой книги могла бы быть весьма печальной;
- моему отцу **Милтиадесу Комвутису** (Miltiades Komvoutis), с ранних лет знакомившему меня с искусством и эстетикой. Если бы не он, я вряд ли бы заинтересовалась дизайном и CSS, и эта книга была бы посвящена чему-нибудь совершенно другому, например C++ или программированию ядра;
- моему дяде и второму отцу **Стратису Веросу** (Stratis Veros) и его чудесной жене **Марии Брере** (Maria Brere), терпевшим мои капризы и невыносимое поведение во время работы над этой книгой. А также их дочерям, **Леони** (Leonie) и **Фиби** (Phoebe), — самым милым девочкам в мире, без которых я закончила бы эту книгу на месяц раньше;
- моей невероятной маме **Марии Веру** (Maria Verou), к сожалению, уже почившей. Все 27 лет, в течение которых мы были в этом мире вместе, она оставалась моим лучшим другом и самой большой поддержкой. Ее жизненная история не может не вдохновлять: она переехала на другой конец света, для того чтобы в 1970-е — во времена, когда большинство женщин в Греции с трудом дотягивали до университета, — стать аспирантом в MIT, и получила степень с отличием. Она привила мне честолюбие, доброту, целостность, независимость и широту взглядов. Но важнее всего то, что она научила меня не относиться к жизни слишком серьезно. Я невероятно скучаю по ней.

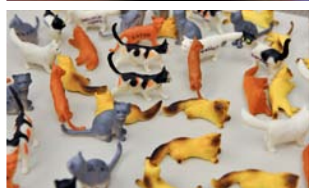
Авторство фотографий

Огромная благодарность людям, публикующим свои фоторaботы со свободными лицензиями Creative Commons; в противном случае во всех примерах в этой книге фигурировали бы фотографии моего кота (на самом деле очень часто так и есть). Вот список фотографий с лицензированием CC, которые я использовала, а также адреса веб-сайтов, где вы их можете найти:



House Made Sausage from Prairie Grass Cafe, Northbrook, Kurman Communications, Inc

http://flickr.com/kurmanphotos/7847424816



Cats that Webchick Is Herding, Kathleen Murtagh

http://flickr.com/ceardach/4549876293



Stone Art, Josef Stuefer

http://flickr.com/josefstuefer/5982121



A Field of Tulips, Roman Boed

http://flickr.com/romanboed/867231576



Resting in the Sunshine, Steve Wilson

http://flickr.com/pokerbrit/10780890983



Naxos Island, Greece, Chris Hutchison

http://flickr.com/employtheskinnyboy/3904743709

Об этой книге

Для кого эта книга

Главная целевая аудитория этой книги — **разработчики CSS среднего и продвинутого уровня**. Избавившись от информации начального уровня, мы можем посвятить больше времени изучению сложных сценариев использования современных возможностей CSS и их разнообразных комбинаций. Это, однако, означает, что я сделала несколько **предположений** относительно вашего уровня подготовки, мой дорогой читатель:

- я предполагаю, что **CSS 2.1 вы знаете назубок** и у вас за плечами несколько лет разработки. Вы не мучаетесь вопросом, как же работает позиционирование. Вы используете генерируемое содержимое для украшения дизайна, не прибегая к помощи лишней разметки или изображений. И вы не развешиваете `!important` по всему коду, так как действительно понимаете специфичности, наследование и каскадирование. Вы знаете составные части блочной модели и вас не способно расстроить схлопывание полей. Вам знакомы разные единицы изменения длины, и вы знаете, в какой ситуации какую из них лучше применить;
- вы достаточно хорошо знакомы с **наиболее популярными возможностями CSS3** — либо из статей, опубликованных в Сети, либо из книг — и пробовали применять их, пусть даже ограничиваясь своими личными проектами. Даже если вы не исследовали их детально, вы знаете, как создать скругленные углы, добавить тень `box-shadow` или определить линейный градиент. Вы уже поиграли с базовыми двумерными трансформациями и немало времени посвятили изучению базовых переходов и анимации;
- у вас есть представление о формате **SVG**, и вы знаете, для чего он используется, даже если файлы в этом формате вы самостоятельно не создаете;
- вы можете читать и понимать **простейший код JavaScript**, такой, например, какой требуется для создания элементов, манипулирования их атрибутами и добавления их в документ;
- вы слышали о **препроцессорах CSS** и знаете, на что они способны, даже если решили в своей работе их не использовать;

- вы не плаваете в **математике уровня средней школы**: квадратные корни, теорема Пифагора, синусы, косинусы и логарифмы.

Однако для того чтобы читатели, не удовлетворяющие вышеперечисленным требованиям, также могли наслаждаться книгой, в начале некоторых секретов я добавила врезку **«Предварительные требования»**, в которой вкратце перечисляю аспекты CSS или предыдущие секреты, с которыми необходимо ознакомиться, чтобы понять и научиться применять текущий секрет. (Сюда, разумеется, не входят возможности CSS 2.1, потому что в этом случае врезка стала бы очень длинной.) Она выглядит так:

ПРЕДВАРИТЕЛЬНЫЕ ТРЕБОВАНИЯ

Знание свойства `box-shadow`, базовое знание градиентов CSS, секрет «Гибкие эллипсы»

Таким образом, даже если какие-то вещи вам неизвестны, вы можете почитать о них, а затем снова вернуться к секрету. **При условии, что предварительные требования соблюдены, секреты можно читать в любом порядке**, хотя имеет смысл все же придерживаться того порядка, в котором они представлены в книге, так как он не случаен и я посвятила достаточно времени обдумыванию последовательности.

Обратите внимание, что я упомянула о «разработчиках CSS» и что «навыки дизайна» среди представленных выше предположений не числятся. Важно помнить, что **это не книга о дизайне**. Хотя она неизбежно затрагивает определенные принципы дизайна и описывает некоторые улучшения взаимодействия с пользователем, «Секреты CSS» — это в первую очередь книга **о решении проблем с кодом**. CSS-код порождает некий визуальный результат, но это все еще код, точно такой же, как код SVG, WebGL/OpenGL или JavaScript Canvas API, а не дизайн. Для написания хорошего гибкого CSS требуется такое же аналитическое мышление, что и для программирования. Сегодня, когда большинство людей используют для своего CSS-кода препроцессоры со всеми их переменными, математикой, условными выражениями и циклами, написание кода CSS уже практически неотличимо от программирования!

Это не означает, что я не приглашаю дизайнеров прочитать эту книгу. Каждый, кто обладает достаточным опытом программирования на CSS, может почерпнуть из нее что-нибудь полезное, и многие талантливые дизайнеры также способны писать превосходный CSS-код. Однако необходимо отметить, что я **не** ставила себе целью в этой книге учить вас совершенствованию визуального дизайна или удобства использования веб-сайта, даже если это и произошло по факту.

Форматирование и условные обозначения

Эта книга состоит из 47 «секретов», тематически сгруппированных в семь глав. Все эти секреты более-менее независимы и — при условии, что все предварительные требования выполняются, — с ними можно знакомиться в любом порядке. Демонстрационные примеры в каждом из секретов — это не готовые веб-сайты и даже не их части. Я намеренно делала их как можно меньше и проще, для того чтобы их было удобнее изучать. Я исхожу из предположения, что вы уже имеете представление о том, что намереваетесь реализовать. Цель этой книги — предоставлять не дизайнерские идеи, а решения по их реализации.



Рис. П.1. Это пример иллюстрации во врезке, главный герой которой — мой котенок сэр Адам Кэтлейс

Каждый секрет разбит на два или более раздела. Первый раздел, озаглавленный «Проблема», содержит описание распространенной проблемы, которую мы будем решать с помощью CSS. Иногда в таком введении я также описываю популярные, но недостаточно хорошие решения данной проблемы (например, решения, требующие объемной разметки, жестко закодированных значений и т. п.) и чаще всего завершаю его одной из вариаций вопроса: существует ли лучший способ реализовать то же самое?

Примечания, подобные этому, содержат дополнительную информацию или объяснение термина, встретившегося в тексте.

За описанием проблемы следует одно или несколько решений. Вдохновением для этой книги послужили семинары по CSS, которые я проводила на различных конференциях, поэтому я постаралась сохранить формат интерактивной презентации, насколько это возможно в книге. Каждому решению сопутствуют несколько иллюстраций, демонстрирующих визуальный результат каждого шага решения в случае, если он приводит к каким-то видимым изменениям. Поскольку иллюстрации не всегда находятся рядом с текстом, описывающим происходящее на них, все иллюстрации пронумерованы и я ссылаюсь на них по номерам. Пример иллюстрации вы можете видеть на рис. П.1, а это предложение — пример упоминания иллюстрации.

! Это предупреждение. Его назначение — предупредить вас (удивительное совпадение, не так ли?) о возможных ошибочных предположениях и вариантах, как что-то может пойти не так.

Строковый код выделяется `моноширинным шрифтом`, а названия и коды цветов часто дополняются небольшой цветовой меткой (например, `#f06`). Блочные фрагменты кода выглядят так:

```
background: url("adamcatlace.jpg");
```

или так:

HTML

```
<figure>
    <img src="adamcatlace.jpg" />
    <figcaption>Sir Adam Catlace</figcaption>
</figure>
```

Как вы заметили, в случае, когда язык во фрагменте кода отличается от CSS, он указан в заголовке листинга. Помимо этого, если в примере задействован только один элемент, без каких-либо псевдоклассов или псевдоэлементов, то обычно для краткости я опускаю в коде селекторы и фигурные скобки (`{}`).

Все примеры на JavaScript в этой книге относятся к простейшему уровню и не требуют никакой инфраструктуры или библиотек. Используется только одна вспомогательная функция, `$$()`, необходимая для того, чтобы было проще проходить по множеству элементов, соответствующих определенному селектору CSS. Вот определение этой функции:

JS

```
function $$(selector, context) {
    context = context || document;
    var elements = context.querySelectorAll(selector);
    return Array.prototype.slice.call(elements);
}
```

ЗАНИМАТЕЛЬНАЯ СТРАНИЧКА.
ВРЕЗКА С ИНТЕРЕСНОЙ ИНФОРМАЦИЕЙ

Врезки, озаглавленные «Занимательная страничка», — это интересные заметки, связанные с обсуждаемой темой, например, историческая или техническая справка о рассматриваемой возможности CSS. Их не обязательно читать, для того чтобы понять или начать использовать основной материал, и все же они могут оказаться полезными любознательному читателю.

К каждому секрету прилагается один или несколько интерактивных примеров, для доступа к которым используются короткие и легкие в запоминании URL-адреса на сайте http://play.csssecrets.io. Ссылки на них выглядят так:

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**polka**

Я настоятельно рекомендую вам внимательно изучать примеры из врезок «Попробуйте сами!», особенно если вы запутались в описании техники или зашли в тупик в попытке воспроизвести пример самостоятельно.

Надлежащая благодарность: если описанная техника впервые была задокументирована другим участником сообщества, я обязательно упоминаю об этом во врезках «Благодарности», подобных этой, указывая также URL-адрес источника. Мы все знаем, что необходимость искать раздел «Список литературы» в конце книги ужасно затрудняет чтение, поэтому я ссылаюсь на источники прямо в контексте.



Благодарности

БУДУЩЕЕ.
БУДУЩИЕ РЕШЕНИЯ

Врезки «Будущее» содержат описание техник, для которых уже подготовлены черновые спецификации, но которые на момент написания этой книги еще не реализованы. Читателю следует всегда проверять, поддерживаются ли эти техники, так как вполне возможна ситуация, что они были реализованы уже после публикации книги. В случаях, когда возможность настолько малоизвестна, что упоминания о ней может не быть даже на веб-сайтах поддержки браузеров, эта врезка включает ссылку на тест, который читатель может загрузить, перейдя по короткому, легко запоминающемуся URL-адресу, такому, как показан ниже в примере «Протестируйте!». Оформление таких тестов обычно включает оттенки зеленого, когда возможность поддерживается, и оттенки красного в противном случае. Точные инструкции приведены в коде в форме комментариев.

ПРОТЕСТИРУЙТЕ!

http://play.csssecrets.io/**test-conic-gradient**

В конце почти всех секретов вы найдете список связанных спецификаций, который выглядит так:

СВЯЗАННЫЕ СПЕЦИФИКАЦИИ

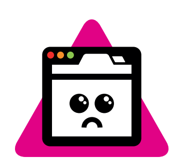
CSS Backgrounds & Borders: http://w3.org/TR/css-backgrounds

Selectors: http://w3.org/TR/selectors

Scalable Vector Graphics: http://w3.org/TR/SVG

Сюда входят ссылки на все спецификации, возможности из которых были упомянуты в секрете. Однако так же как и врезка «Предварительные требования», врезка «Связанные спецификации» не включает функциональность **CSS 2.1** (http://w3.org/TR/CSS21), иначе одни и те же возможности пришлось бы перечислять после каждого секрета. Это означает, что те несколько секретов, в которых мы обсуждаем только возможности из CSS 2.1, не дополняются врезкой «Связанные спецификации».

Поддержка браузерами и резервные решения



Ограниченная поддержка

Вероятно, самая необычная особенность этой книги — **полное отсутствие таблиц совместимости с браузерами**. Это осознанное решение, так как с учетом сегодняшних релизных циклов браузеров подобная информация неизбежно устаревает еще до того, как книга успевает попасть на полки магазинов. Я считаю, что **неточная информация о поддержке браузерами вводит разработчика в заблуждение, что еще хуже, чем отсутствие такой информации**.

Однако большинство представленных здесь секретов либо вполне прилично поддерживаются браузерами, либо для них существуют хорошие резервные решения. Если поддержка браузерами определенной техники находится на слишком низком уровне, вы увидите предупреждающий значок «Ограниченная поддержка» рядом с заголовком соответствующего решения, как здесь, рядом с этим абзацем. Этого должно быть достаточно, для того чтобы вы поняли: не стоит использовать данное решение, не проверив уровень поддержки браузерами, а также не позаботившись о качественных резервных решениях.

В Сети вы найдете множество превосходных веб-сайтов, предлагающих самую свежую информацию о поддержке браузерами. Вот несколько из них:

- **Can I Use...?** (http://caniuse.com)
- **WebPlatform.org**
- **Mozilla Developer Network** (http://developer.mozilla.org)
- статья в **Wikipedia** *Comparison of Layout Engines (Cascading Style Sheets)* (http://en.wikipedia.org/wiki/Comparison_of_layout_engines_(Cascading_Style_Sheets))

Иногда вы будете обнаруживать, что определенная возможность поддерживается, но ее реализация в разных браузерах немного отличается. Например, она может требовать **браузерного префикса** или ее **синтаксис может быть несколько иным**. В примерах в этой книге я использую только синтаксис без префиксов, как он определяется в стандартах. Однако практически в любой ситуации вы можете одновременно использовать разные варианты синтаксиса, и тогда нужный будет выбираться автоматически в соответствии с правилами каскадирования. По этой причине **стандартную версию всегда следует указывать последней**. Например, для получения вертикального линейного градиента от цвета `yellow` до `red` в книге я всегда буду использовать только стандартную версию:

```
background: linear-gradient(90deg, yellow, red);
```

Однако если вы хотите обеспечить поддержку очень старых браузеров, возможно, в итоге ваш код будет выглядеть примерно так:

```
background: -moz-linear-gradient(0deg, yellow, red);
background: -o-linear-gradient(0deg, yellow, red);
background: -webkit-linear-gradient(0deg, yellow, red);
background: linear-gradient(90deg, yellow, red);
```

Поскольку ландшафт этих различий настолько же нестабилен, как и поддержка браузерами, я ожидаю, что в вашей работе проверка таких вещей будет одним из этапов обязательного исследования перед применением той или иной возможности CSS, поэтому не обсуждаю их в решениях, представленных в данной книге.

Подробнее о браузерных префиксах, почему они существуют и как абстрагироваться от них в вашем коде, вы можете прочитать в разделе «Песнь льда, пламени и браузерных префиксов».

Аналогично, хорошей практикой считается обеспечение резервных решений, для того чтобы ваши веб-сайты не ломались в старых браузерах, пусть даже ценой более простецкого внешнего вида. Когда обходные решения очевидны, я не заостряю на них внимание, так как предполагаю, что вы знакомы с принципами каскадирования. Например, при определении градиента, скажем, такого, как показанный выше, вы могли бы добавить в самом начале сплошной цвет.

При выборе такого цвета рекомендуется останавливаться на среднем из двух цветов градиента (в данном случае `rgb(255, 128, 0)`):

```
background: rgb(255, 128, 0);
background: -moz-linear-gradient(0deg, yellow, red);
background: -o-linear-gradient(0deg, yellow, red);
background: -webkit-linear-gradient(0deg, yellow, red);
background: linear-gradient(90deg, yellow, red);
```

Однако иногда каскадирование не позволяет обеспечить надежное резервное решение. Тогда в качестве последнего средства можно прибегнуть к помощи инструментов, подобных **Modernizr** (http://modernizr.com), которые добавляют классы вроде `textshadow` или `no-textshadow` к корневому элементу (`<html>`), чтобы вы могли с помощью них **обращаться к элементам только в том случае, когда нужные возможности действительно поддерживаются (или не поддерживаются)**, например:

```
h1 { color: gray; }

.textshadow h1 {
    color: transparent;
    text-shadow: 0 0 .3em gray;
}
```

Если возможность, для которой вы пытаетесь создать резервное решение, достаточно новая, то можно использовать правило `@supports`, «родное» для Modernizr. Например, предыдущий фрагмент кода превратится в такой:

```
h1 { color: gray; }

@supports (text-shadow: 0 0 .3em gray) {
    h1 {
        color: transparent;
        text-shadow: 0 0 .3em gray;
    }
}
```

Однако к использованию `@supports` следует подходить с большой осторожностью. Применив его здесь, мы ограничили описываемый эффект не просто браузерами, поддерживающими тени для текста, но и браузерами, поддерживающими дополнительно правило `@supports`, а это куда более ограниченное множество.

И наконец, всегда есть вариант добавить несколько строк кода JavaScript ручной работы, который будет определять, поддерживается ли возможность, и добавлять классы в корневой элемент так же, как это делает Modernizr. Основной

способ, как проверить, поддерживается ли свойство, — посмотреть, существует ли он, воспользовавшись объектом `element.style` любого элемента:

JS

```
var root = document.documentElement; // <html>

if ('textShadow' in root.style) {
    root.classList.add('textshadow');
}
else {
    root.classList.add('no-textshadow');
}
```

Если нам нужно проверить несколько свойств, предыдущую проверку легко превратить в функцию:

JS

```
function testProperty(property) {
    var root = document.documentElement;

if (property in root.style) {
    root.classList.add(property.toLowerCase());
    return true;
}

root.classList.add('no-' + property.toLowerCase());
    return false;
}
```

Для того чтобы протестировать значение, нужно присвоить его свойству и проверить, сохранит ли его браузер. Поскольку здесь мы модифицируем стили, а не просто проверяем их существование, в тесте разумно использовать элемент-заглушку:

JS

```
var dummy = document.createElement('p');
dummy.style.backgroundImage = 'linear-gradient(red,tan)';

if (dummy.style.backgroundImage) {
    root.classList.add('lineargradients');
}
else {
    root.classList.add('no-lineargradients');
}
```

Это также легко преобразуется в функцию:

JS

```
function testValue(id, value, property) {
    var dummy = document.createElement('p');
    dummy.style[property] = value;

    if (dummy.style[property]) {
        root.classList.add(id);
        return true;
    }

    root.classList.add('no-' + id);
    return false;
}
```

Тестирование селекторов и `@rules` немного сложнее, но выполняется по тому же принципу: когда дело доходит до CSS, браузеры отбрасывают все, что они не понимают, так что для проверки того, была ли возможность распознана, можно динамично применить ее и посмотреть, сохранил ли ее браузер. Необходимо всегда помнить, однако, что даже если браузер в состоянии **разобрать синтаксис** возможности CSS, это **не гарантирует, что таковая возможность реализована правильно и что она вообще реализована в принципе**.

1

Введение

Веб-стандарты: свои или чужие?

Процесс подготовки стандартов



Рис. 1.1. «Стандарты — как сосиски: лучше не видеть, как они делаются» (анонимный участник рабочей группы W3C)

Вопреки распространенному мнению, **W3C (World Wide Web Consortium, Консорциум Всемирной паутины) не «делает» стандарты**. На самом деле он играет роль форума, помогая заинтересованным сторонам собираться в так называемые рабочие группы W3C (W3C Working Groups) и проводить необходимую подготовительную работу. Разумеется, сам W3C также не остается простым наблюдателем: он устанавливает основные правила и контролирует процесс. Тем не менее **фактическим написанием спецификаций занимаются (в основном) другие люди, а не сотрудники W3C**.

Спецификации CSS, в частности, пишутся членами рабочей группы по каскадным таблицам стилей CSS — CSS Working Group, которую для краткости часто называют просто рабочей группой CSS, или CSS WG. На момент написания этой главы рабочая группа CSS состоит из 98 участников, в том числе:

- **86** сотрудников компаний-участниц W3C (88%);
- **7** приглашенных экспертов, включая вашу покорную слугу (7%);
- **5** штатных сотрудников W3C (5%).

Как вы могли заметить, большинство членов рабочей группы (88%) — сотрудники *компаний-участниц W3C*. Это организации, такие как производители браузеров, владельцы популярных веб-сайтов, исследовательские институты, общетехнологические компании и т. д., лично заинтересованные в процветании веб-стандартов. Их ежегодные членские взносы обеспечивают большую часть финансирования W3C, что позволяет Консорциуму распространять свои

спецификации **бесплатно** и **открыто**, в отличие от других органов по стандартизации, которым приходится взимать за них плату.

Приглашенные эксперты — это веб-разработчики, которых попросили принять участие в процессе подготовки стандартов. Они основательно подкованы с технической стороны вопроса, продемонстрировали свою приверженность сообществу и на протяжении длительного времени оказывают помощь его членам.

И наконец, *штатные сотрудники W3C* — это люди, фактически работающие в Консорциуме, которые содействуют обмену информацией между рабочей группой и W3C.

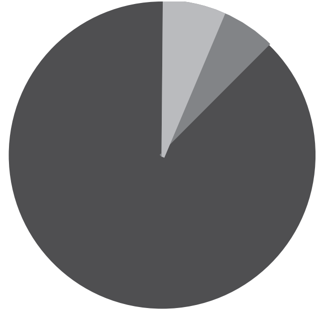


Рис. 1.2. Состав CSS WG:
- компании-участницы
- приглашенные эксперты
- штатные сотрудники W3C

Среди веб-разработчиков широко бытует ошибочное представление о том, что W3C спускает сверху некие стандарты, к которым бедным браузерам приходится подстраиваться, нравится им это или нет. Однако сильнее заблуждаться попросту невозможно: что касается стандартов, **мнение производителей браузеров имеет куда больший вес, чем пожелания W3C**, и это убедительно доказывают цифры, приведенные ниже.

Также в противоположность распространенному мнению **стандарты создаются не в вакууме**, не за закрытыми дверьми. Рабочая группа CSS считает обеспечение прозрачности одной из важнейших своих задач; все ее коммуникации полностью открыты для публики, и каждый свободен высказывать свое мнение и участвовать в обсуждениях:

- большая часть дискуссий ведется в рамках **списка рассылки www-style** (http://lists.w3.org/Archives/Public/www-style). Архивы обсуждений группы общедоступны, и она открыта для каждого желающего присоединиться;
- каждую неделю проводится **телефонная конференция** продолжительностью один час. К участию в ней допускаются только члены рабочей группы, однако протокол конференции в режиме реального времени публикуется на канале **#css** на **IRC-сервере W3C** (http://irc.w3.org/). По завершении конференции протокол приводится в порядок и через несколько дней публикуется в списке рассылки;
- также проводится **ежеквартальное «живое» собрание**, протокол которого ведется и публикуется таким же образом, как для телефонных конференций. Эти собрания открыты для **наблюдения** (аудита), однако для того, чтобы присутствовать на них, требуется запросить специальное разрешение у *председателей* рабочей группы.

Хотели бы узнать больше? Элика Этемад (Elika Etemad), также известная как fantasai, написала **серию поразительных статей о деятельности рабочей группы CSS** (http://fantasai.inkedblade.net/weblog/2011/inside-csswg). Горячо рекомендуется к прочтению.

Все это составляет часть рабочего процесса W3C и связано с принятием решений. Однако люди, несущие реальную ответственность за то, чтобы облекать данные решения в письменную форму (то есть за фактическую разработку спецификаций), — это *редакторы спецификаций* (Spec Editor). Редакторами спецификаций могут быть штатные сотрудники W3C, разработчики браузеров, заинтересованные приглашенные эксперты или сотрудники компаний-участниц, для которых это основная работа на полную ставку — компании платят им для того, чтобы совершенствовать и продвигать стандарты для всеобщего блага.

Каждая спецификация проходит несколько этапов на своем пути от зарождения до окончательной зрелости.

1. **Редакторский черновик (Editor's Draft, ED):** на первой стадии разработки спецификация может быть простым наброском идей редактора. К этой стадии не предъявляются никакие требования, и нет никаких гарантий, что данная версия будет утверждена рабочей группой. Тем не менее любая ревизия обязательно проходит эту стадию: все изменения сначала вносятся в форме редакторского черновика и только после этого публикуются.
2. **Первый публичный рабочий черновик (First Public Working Draft, FPWD):** первая опубликованная версия спецификации, которую рабочая группа считает готовой для представления аудитории с целью сбора отзывов и общественного мнения.
3. **Рабочий черновик (Working Draft, WD):** за первым рабочим черновиком следует еще множество. Каждый содержит очередные улучшения, основанные на отзывах рабочей группы и широкого сообщества. Зачастую первые реализации начинают создаваться именно на этой стадии, хотя нет ничего необычного также и в появлении экспериментальных реализаций на более ранних этапах подготовки спецификаций.



Рис. 1.3. Частенько говорят, что председательствовать в рабочей группе W3C — это то же самое, что пасти котов

4. **Рекомендованный кандидат (Candidate Recommendation, CR):** данная версия считается относительно стабильной. Это означает, что пришло время для реализаций и тестирования. Спецификация не может перейти на следующую стадию без подготовки полного набора тестов и создания как минимум двух независимых реализаций.

5. **Предложенная рекомендация (Proposed Recommendation, PR):** последний шанс для компаний-участниц W3C выразить свое несогласие со спецификацией. Такое случается редко, поэтому чаще всего переход каждой PR-спецификации на следующий, финальный этап — всего лишь вопрос времени.

6. **Рекомендация (Recommendation, REC):** финальная стадия подготовки спецификации W3C.

Один или два члена рабочей группы также исполняют роль *председателей*. Председатели несут ответственность за организацию собраний, координирование звонков, контроль над хронометражем и, в целом, за общее управление всем процессом. Выполнение обязанностей председателя отнимает огромное количество сил и времени, и зачастую эту деятельность сравнивают с **пастьбой котов**. Разумеется, каждый, кто знаком с процессом подготовки стандартов, знает, что это сравнение весьма сомнительно, — ведь пасти котов куда проще.

CSS3, CSS4 и прочие мифические чудовища

Спецификация **CSS 1** была очень короткой и относительно простой. Ее опубликовали в 1996 году Хокон Виум Ли и Берт Бос. Она была так компактна, что ее целиком сверстали на одной HTML-странице, а для печати спецификации требовалось около 68 листов бумаги формата A4.

Опубликованная в 1998 году спецификация **CSS 2** была более строго определенной и влиятельной, и в ее подготовке принимали участие еще два редактора спецификаций: Крис Лилли и Иан Джейкобс. На данном этапе размер спецификации достиг 480 (!) печатных страниц, и она была уже слишком велика, для того чтобы человек мог полностью уместить ее в памяти.

После CSS второго уровня рабочая группа CSS пришла к осознанию, что язык становится слишком велик для одной спецификации. И дело не только в том, что документ стал чрезмерно громоздким для чтения и редактирования, — единая спецификация задерживала развитие CSS. Вспомните, что для достижения спецификацией финального этапа каждая содержащаяся в ней возможность должна получить по меньшей мере две независимые реализации и обязана быть подвергнута тщательнейшему тестированию. Это уже становилось

непрактичным и нецелесообразным. Таким образом, было принято решение, что для того, чтобы продолжать двигаться вперед, общую спецификацию CSS нужно разбить на множество отдельных спецификаций (модулей), каждый с собственным версионированием. Модули, расширяющие возможности, которые уже присутствовали в CSS 2.1, переводились на уровень 3. Примеры таких модулей приведены далее:

- **CSS Syntax** (http://w3.org/TR/css-syntax-3)
- **CSS Cascading and Inheritance** (http://w3.org/TR/css-cascade-3)
- **CSS Color** (http://3.org/TR/css3-color)
- **Selectors** (http://w3.org/TR/selectors)
- **CSS Backgrounds & Borders** (http://w3.org/TR/css3-background)
- **CSS Values and Units** (http://w3.org/TR/css-values-3)
- **CSS Text** (http://w3.org/TR/css-text-3)
- **CSS Text Decoration** (http://w3.org/TR/css-text-decor-3)
- **CSS Fonts** (http://w3.org/TR/css3-fonts)
- **CSS Basic User Interface** (http://w3.org/TR/css3-ui)

Однако модули, посредством которых вводились совершенно новые концепции, начинали свою историю с уровня 1. Вот несколько примеров:

- **CSS Transforms** (http://w3.org/TR/css-transforms-1)
- **Compositing and Blending** (http://w3.org/TR/compositing-1)
- **Filter Effects** (http://w3.org/TR/filter-effects-1)
- **CSS Masking** (http://w3.org/TR/css-masking-1)
- **CSS Flexible Box Layout** (http://w3.org/TR/css-flexbox-1)
- **CSS Grid Layout** (http://w3.org/TR/css-grid-1)

Несмотря на популярность модного термина «CSS3», конкретной спецификации, определяющей нечто подобное, в действительности не существует — в отличие, например, от спецификации для CSS 2.1 и ее предшественников. Употребляя это слово, чаще всего авторы имеют в виду некий произвольный набор спецификаций уровня 3 плюс несколько спецификаций первого уровня. Несмотря на то что разработчиками достигнута определенная степень консенсуса относительно того, какие спецификации входят в «CSS3», с годами, с учетом разной скорости проработки и развития разных модулей CSS, будет становиться все сложнее использовать такие обозначения, как CSS3, CSS4 и т. д., не вводя читателей в заблуждение.

Песнь льда, пламени и браузерных префиксов

Разработка стандартов всегда по своей природе парадоксальна: группы по подготовке стандартов нуждаются в информации от разработчиков, для того чтобы создавать спецификации, регулирующие реальные потребности разработчиков. Но разработчики, в целом, не слишком заинтересованы в том, чтобы тестировать вещи, которые они не могут применять в своей работе. Когда экспериментальные технологии начинают широко использоваться в производстве, у рабочей группы не остается иного выхода, кроме как придерживаться ранней, экспериментальной версии технологии, так как ее изменение способно поломать уже существующие веб-сайты. Очевидно, это полностью сводит на нет преимущества, которые способно принести тестирование ранних версий стандартов реальными разработчиками.

За прошедшие годы было предложено множество вариантов выхода из этой непростой ситуации, но все они далеки от идеала. Повсеместно презираемые браузерные префиксы — один из них. Идея заключалась в том, что для каждого браузера могут быть реализованы экспериментальные (или даже патентованные) возможности, к названиям которых необходимо добавлять специальный префикс. Наиболее распространенные префиксы — это `-moz-` для Firefox, `-ms-` для IE, `-o-` для Opera и `-webkit-` для Safari и Chrome. Разработчикам предлагалось свободно экспериментировать с этими специальными возможностями и делиться своими впечатлениями с рабочей группой. Рабочая группа, в свою очередь, должна была учитывать обратную связь от разработчиков при подготовке спецификаций, постепенно доводя соответствующую функциональность до совершенства. Так как у финальной, стандартизированной версии должно было быть другое название (без префикса), ее добавление не должно было порождать коллизии в продуктах, использующих уже существующие, обремененные префиксом эквиваленты.

Звучит отлично, не так ли? Но, как вы уже, вероятно, знаете, реальность оказалась совершенно непохожей на то, что планировалось воплотить. Когда разработчики осознали, что эти экспериментальные зависимые от браузера свойства позволяют с легкостью создавать эффекты, реализация которых ранее требовала огромных усилий и запутанных обходных путей, они принялись использовать их где только можно. Свойства с браузерными префиксами быстро превратились в модную тенденцию в мире CSS. Выпускались учебники, публиковались ответы на сайте StackOverflow, и скоро практически каждый уважающий себя CSS-разработчик обвешивал ими свои сайты с ног до головы.

В конце концов разработчики осознали, что если использовать только существующие браузерные префиксы, то к уже имеющемуся коду необходимо возвращаться и добавлять новые объявления каждый раз, когда в новом браузере появляется поддержка их любимой классной возможности CSS. Не говоря уж

о том, что все префиксы, необходимые для той или иной возможности, вообще довольно сложно держать в памяти. Решение? Конечно же, всегда использовать все возможные браузерные префиксы, в конце заодно добавляя версию без префикса, для того чтобы гарантировать правильную обработку кода в будущем. В результате код стал выглядеть примерно так:

```
-moz-border-radius: 10px;
-ms-border-radius: 10px;
-o-border-radius: 10px;
-webkit-border-radius: 10px;
border-radius: 10px;
```

Среди этих объявлений два избыточны: `-ms-border-radius` и `-o-border-radius` никогда ни в каком браузере не существовали, так как в IE и Opera с самого начала было реализовано свойство `border-radius` безо всякого префикса.

Очевидно, что повторять каждое объявление до пяти раз невероятно утомительно, а результирующий код не приспособлен для нормальной поддержки. Появление инструментов, которые автоматизировали бы это, было исключительно вопросом времени:

- на таких веб-сайтах, как **CSS3, Please!** (http://css3please.com) и **pleeease** (http://pleeease.io/playground.html), вы можете вставить CSS-код без префиксов и получить обратно CSS со всеми необходимыми префиксами. Подобные приложения стали одними из первых реализаций автоматического добавления браузерных префиксов, но быстро растеряли свою популярность, так как по сравнению с другими решениями довольно неудобны в использовании;
- **Autoprefixer** (http://github.com/ai/autoprefixer) использует базу данных из **Can I Use...** (http://caniuse.com) для определения, какие префиксы необходимо добавить к коду без браузерных префиксов, и компилирует его локально, как препроцессор;
- моя собственная утилита **-prefix-free** (http://leaverou.github.io/prefixfree) выполняет тестирование возможностей в браузере, определяя, какие префиксы требуются. Ее преимущество в том, что она крайне редко требует обновления, так как получает всю необходимую информацию, включая список свойств, из окружения браузера;
- такие препроцессоры, как **LESS** (http://lesscss.org) и **Sass** (http://sass-lang.com), не предлагают стандартной функциональности добавления префиксов, но многие разработчики создают собственные подборки для возможностей, с которыми они чаще всего используют браузерные префиксы, и в обращении можно найти несколько подобных библиотек.

Поскольку разработчики использовали версии без префиксов в качестве гарантии работоспособности своего кода в будущем, изменять их стало невозможно.

По сути, мы зашли в тупик с полусырыми ранними спецификациями, допускающими лишь незначительные изменения. Совсем скоро все пришли к осознанию, что **браузерные префиксы были эпическим провалом**.

Сегодня браузерные префиксы применяются для новых экспериментальных реализаций очень редко. Вместо этого экспериментальные возможности включаются с помощью конфигурационных флагов, что предотвращает использование их разработчиками в производственном окружении, — ведь вы не можете заставлять пользователей менять локальные настройки для того, чтобы веб-сайт на их машинах отображался правильно. Разумеется, следствием этого стало то, что теперь гораздо меньше разработчиков тестируют экспериментальные возможности, но мы все же получаем достаточно обратной связи — и, возможно, более высококачественной обратной связи, — не испытывая при этом сложностей, порождаемых браузерными префиксами. Однако пройдет еще немало времени, прежде чем мы полностью избавимся от неприятных последствий существования браузерных префиксов.

Советы по написанию CSS-кода

Минимизируйте дублирование кода

Соответствие кода принципу DRY и обеспечение удобства в сопровождении — одна из самых больших сложностей в разработке программного обеспечения, и это также применимо к CSS. На практике одна из главных составляющих сопровождаемости кода — это **минимизация объема редактирования, необходимого для внесения изменений**. Например, если для того, чтобы увеличить кнопку, вам нужно сделать 10 исправлений во множестве различных правил, велик шанс, что несколько вы все же пропустите, особенно если изначально этот код написан не вами. И даже если исправления очевидны или вы в итоге находите все места, требующие правки, все равно это означает потерю времени, которое можно было бы использовать с большим толком.

Но речь не только о будущих исправлениях. Гибкий CSS означает, что вы пишете CSS-код один раз, а затем с легкостью создаете различные его варианты, исправляя лишь небольшой объем кода, так как переопределения требуют всего несколько значений. Давайте рассмотрим пример.

Взгляните на следующий CSS-код, определяющий стиль кнопки, показанной на рис. 1.4:

```
padding: 6px 16px;
border: 1px solid #446d88;
background: #58a linear-gradient(#77a0bb, #58a);
border-radius: 4px;
box-shadow: 0 1px 5px gray;
color: white;
text-shadow: 0 -1px 1px #335166;
font-size: 20px;
line-height: 30px;
```

Этот код порождает несколько проблем сопровождаемости, но в наших силах от них избавиться. Первое, что бросается в глаза, — это параметры шрифта.

Если мы решим изменить размер шрифта (например, создать вариацию для более важных и крупных кнопок), то нам придется также исправить интервал между строками, так как оба этих параметра заданы абсолютными значениями. Помимо этого, интервал между строками никак не отражает его взаимосвязи с размером шрифта, поэтому в случае необходимости использовать шрифт иного размера нам придется делать вычисления, подбирая подходящий интервал. **Когда значения зависят друг от друга, старайтесь отражать эти взаимозависимости в коде.** В данном случае интервал между строками составляет 150% размера шрифта. Следовательно, было бы намного удобнее явно показать это в коде:



Рис. 1.4. Кнопка, которую мы будем использовать в нашем примере

```
font-size: 20px;
line-height: 1.5;
```

И раз уж мы все равно здесь, давайте разберемся, почему мы задали размер шрифта абсолютным значением? Определенно, с абсолютными значениями удобно работать, но они втыкают вам нож в спину каждый раз, когда код требует хотя бы малейших изменений. Сейчас, если вы решите сделать размер родительского шрифта больше, вам потребуется поменять каждое правило в таблице стилей, в котором для шрифтов задаются абсолютные параметры. Гораздо лучше использовать проценты или единицы длины `em`:

```
font-size: 125%; /* Предполагаем, что размер родительского шрифта 16px */
line-height: 1.5;
```

Теперь, если мы поменяем размер родительского шрифта, кнопка моментально увеличится. Однако выглядеть она будет по-другому (рис. 1.5), так как все остальные эффекты были подогнаны под кнопку меньшего размера и не масштабируются. Мы можем сделать масштабируемыми также и другие эффекты, задавая все значения в `em`, чтобы они все зависели от размера шрифта. Таким образом, мы будем в состоянии управлять размером кнопки из одного места:

```
padding: .3em .8em;
border: 1px solid #446d88;
background: #58a linear-gradient(#77a0bb, #58a);
border-radius: .2em;
box-shadow: 0 .05em .25em gray;
color: white;
text-shadow: 0 -.05em .05em #335166;
font-size: 125%;
line-height: 1.5;
```



Рис. 1.5. Увеличение размера шрифта ломает другие эффекты, применяемые к кнопке (заметнее всего это со скруглением углов), так как они заданы в абсолютных значениях



Рис. 1.6. Теперь мы можем сделать кнопку больше, и эффекты будут масштабироваться вместе с ней

Теперь наша крупная кнопка больше похожа на масштабированную версию оригинала (рис. 1.6). Обратите внимание, что для некоторых параметров мы оставили абсолютные значения. **Какие эффекты должны масштабироваться с изменением размера кнопки, а какие оставаться прежними — решать вам, это дело вкуса.** В данном примере мы хотим, чтобы толщина рамки оставалась равной `1px` независимо от габаритов кнопки.

Здесь мы хотим, чтобы размер шрифта и прочие параметры определялись относительно размера родительского шрифта, поэтому используем `em`. В некоторых случаях нужно, чтобы они задавались относительно размера корневого шрифта (то есть размера шрифта в `<html>`), и тогда использование `em` приводит к сложным вычислениям. В таких ситуациях можно применять единицы длины `rem`. Относительность — важное свойство в CSS, но всегда необходимо **думать**, относительно **чего** вы задаете те или иные значения.

Но изменение размера кнопки в большую или меньшую сторону не единственное, что мы можем захотеть с ней сделать. Часто возникает необходимость изменить цвет кнопок. Например, мы хотим создать красную кнопку Cancel или зеленую кнопку OK. В текущей реализации нам потребовалось бы для этого переопределить четыре объявления (`border-color`, `background`, `box-shadow`, `text-shadow`), не говоря уже о трудностях с пересчетом разнообразных, более темных или светлых вариаций нашего главного цвета, `#58a` и гадании, насколько светлее или темнее исходного оттенка должен быть каждый из них. А что, если мы захотим поместить нашу кнопку на другой фон, отличный от белого? Серый (`gray`) в качестве цвета тени хорошо смотрится только на белом фоне.

Мы можем без труда избавиться от подобных сложностей, накладывая на главный цвет полупрозрачный белый и черный для получения более светлых или более темных вариаций соответственно:

```
padding: .3em .8em;
border: 1px solid rgba(0,0,0,.1);
background: #58a linear-gradient(hsla
            (0,0%,100%,.2), transparent);
border-radius: .2em;
box-shadow: 0 .05em .25em rgba(0,0,0,.5);
color: white;
text-shadow: 0 -.05em .05em rgba(0,0,0,.5);
font-size: 125%;
line-height: 1.5;
```

СОВЕТ

Используйте HSLA вместо RGBA для полупрозрачного белого — там меньше символов и нет повторений, что позволяет быстрее набирать код.

Теперь для создания вариаций с разными цветами нам нужно всего лишь переопределить `background-color` (рис. 1.7):

```
button.cancel {
    background-color: #c00;
}

button.ok {
    background-color: #6b0;
}
```



Рис. 1.7. Все, что нам потребовалось для создания этих цветных вариаций, — изменить цвет фона

Наша кнопка уже стала намного более гибкой. Но этот пример не демонстрирует всех возможностей реализации в вашем коде принципов DRY. В следующих разделах вы найдете еще несколько полезных советов.

Сопровождаемость или лаконичность

Иногда сопровождаемость и лаконичность кода становятся взаимоисключающими свойствами. Даже в предыдущем примере финальная версия кода оказалась немного длиннее исходной. Рассмотрим следующий фрагмент, предназначенный для создания рамки толщиной 10px с каждой стороны элемента, **за исключением левой:**

```
border-width: 10px 10px 10px 0;
```

Здесь всего одно объявление, но для того, чтобы изменить толщину рамки, нам пришлось бы сделать три исправления. Гораздо проще было бы редактировать этот код, если бы в нем было два объявления, и, возможно, в этом случае он также стал бы читабельнее:

```
border-width: 10px;
border-left-width: 0;
```

currentColor

В **CSS уровня 3** (http://w3.org/TR/css3-color) у нас появилось много новых ключевых слов для обозначения цвета, таких как `lightgoldenrodyellow`, польза которых сомнительна. Однако для работы с цветом нам также дали особое новое ключевое слово, позаимствованное из SVG: `currentColor`. Ему не соответствует никакое статичное значение цвета. В действительности оно всегда разрешается в значение свойства `color`, что, по сути, делает его **первой когда-либо существовавшей переменной в CSS**. Переменной с очень ограниченными возможностями, но все же переменной.

Например, предположим, что мы хотим, чтобы все горизонтальные разделители (все элементы `<hr>`) автоматически окрашивались в цвет текста. Благодаря `currentColor` мы можем реализовать это следующим образом:

Кто-то может утверждать, что на самом деле первой переменной в CSS была единица длины `em`, ссылающаяся на значение `font-size`. Большинство процентных значений играют схожую роль, хотя и выглядят куда менее вдохновляюще.

```
hr {
    height: .5em;
    background: currentColor;
}
```

Возможно, вы заметите, что многие существующие свойства ведут себя очень похоже. Например, если при описании рамки не задавать цвет, то она автоматически получает цвет текста. Причина в том, что `currentColor` также служит исходным значением многих цветовых свойств CSS: `border-color`, цветов `text-shadow` и `box-shadow`, `outline-color` и др.

В будущем, когда у нас появятся функции для управления цветами в чистом CSS, `currentColor` станет еще полезнее, так как мы сможем использовать различные вариации этого значения.

Наследование

Хотя большинство разработчиков знают о существовании ключевого слова `inherit`, о нем часто забывают. Ключевое слово `inherit` можно использовать в любом свойстве **CSS**, и оно всегда соответствует вычисленному значению родительского элемента (для псевдоэлементов это элемент, для которого они были сгенерированы). Например, вы хотите, чтобы в элементах формы использовался тот же шрифт, что и для всей остальной страницы. Для этого не нужно указывать его еще раз, просто используйте `inherit`:

```
input, select, button { font: inherit; }
```

Аналогично, можно с помощью `inherit` окрасить гиперссылки в тот же цвет, что и остальной текст:

```
a { color: inherit; }
```

Ключевое слово `inherit` также часто бывает удобно использовать для оформления фона. Например, чтобы создавать выноски, в которых указатель автоматически наследует фон и рамку (рис. 1.8):

Your username:
leaverou
Only letters, numbers, underscores (_) and hyphens (-) allowed!

Рис. 1.8. Выноска, на которой указатель наследует цвет фона и рамку от родительского элемента

```
.callout { position: relative; }

.callout::before {
    content: "";
    position: absolute;
    top: -.4em; left: 1em;
    padding: .35em;
    background: inherit;
    border: inherit;
    border-right: 0;
    border-bottom: 0;
    transform: rotate(45deg);
}
```

Доверяйте глазам, а не числам

Человеческий глаз — далеко не идеальное устройство ввода. Иногда точные измерения приводят к результатам, которые кажутся нам неаккуратными, и это необходимо учитывать при разработке дизайна. Например, в литературе, посвященной визуальному дизайну, всегда особо подчеркивается тот факт, что наши глаза не способны правильно воспринимать объекты, выровненные по вертикали. Для того чтобы создать впечатление, что объект центрирован по вертикали, его необходимо поместить чуть выше геометрической середины. Вы можете убедиться в существовании данного явления, посмотрев на рис. 1.9.



Рис. 1.9. На первом прямоугольнике коричневый квадратик математически выровнен по вертикали, но нам так не кажется. На втором прямоугольнике квадратик находится чуть выше геометрического центра, но человеческий глаз видит его так, словно он находится точно в центре

Схожая особенность существует и в дизайне шрифтов: круглые символы, такие как «O», должны быть немного крупнее прямоугольных, так как наше восприятие круглых форм искажено и нам кажется, что они меньше, чем на самом деле. Вы найдете подтверждающий это пример на рис. 1.10.



Рис. 1.10. Кажется, что круг меньше, но ограничивающая его рамка по размеру совпадает с квадратом

Подобные **оптические иллюзии очень распространены в любых формах визуального дизайна**, и вы не должны забывать о них. Самый известный пример — пустые поля в контейнерах с текстом. Эта проблема возникает всегда, независимо от объема текста, — контейнер может содержать как одно слово, так и несколько абзацев. Если задать поля одинакового размера по всем четырем сторонам контейнера, то в результате они будут казаться нам



Рис. 1.11. Поля одинакового размера (здесь: `.5em`) по всем четырем сторонам контейнера с текстом создают впечатление, что сверху и снизу пустого пространства больше



Рис. 1.12. Мы определили более широкие поля (здесь: `.3em .7em`) слева и справа, и теперь контейнер с текстом выглядит намного более сбалансированным

неодинаковыми, как видно на рис. 1.11. Причина в том, что **буквы вытянуты по вертикали, а сверху и снизу скруглены**, и наши глаза воспринимают это дополнительное пространство как дополнительное поле. Следовательно, **сверху и снизу необходимо задавать поля меньшего размера**, для того чтобы в итоге поля со всех сторон выглядели одинаковыми. Это различие иллюстрирует рис. 1.12.

Об отзывчивом веб-дизайне

Отзывчивый веб-дизайн (Responsive Web Design, RWD) остается повальным увлечением вот уже несколько лет. Однако ударение зачастую делается на том, как важно, чтобы веб-сайты были «отзывчивыми», а о множестве других преимуществ, которые обеспечивает следование принципам RWD, умалчивается.

Распространенная практика заключается в том, что веб-сайт тестируется со многими вариантами разрешения экрана, а возникающие проблемы решаются путем добавления все новых и новых медиазапросов. Однако **каждый медиазапрос делает CSS-код еще более сложным для внесения будущих правок**, поэтому к их созданию не следует относиться легкомысленно. Каждая будущая правка в CSS-коде требует проверки, зависит ли от нее срабатывание медиазапросов и, возможно, также обновление этих медиазапросов. Об этом часто забывают, что приводит к авариям. Чем больше медиазапросов вы добавляете, тем более хрупким становится ваш CSS-код.

Но я не говорю, что использование медиазапросов — однозначно плохая практика. **При правильном применении они могут быть незаменимыми.** Однако прибегать к ним следует лишь как к последнему средству, после того как все остальные попытки сделать дизайн веб-сайта гибким провалились, или же если мы намереваемся полностью менять определенный аспект дизайна при отображении на экране меньшего/большего размера (например, делать боковую врезку горизонтальной). Суть в том, что

СОВЕТ

Попробуйте в своих медиазапросах вместо пикселов использовать единицы длины `em`. Это позволит сделать так, чтобы изменения макета инициировались также при масштабировании текста.

медиазапросы не обеспечивают постоянства исправления ошибок. Они определяют своего рода пороги (также известные как точки прерывания), и если весь остальной код написан так, чтобы обеспечивать гибкость продукта, то медиазапросы будут лишь исправлять результат для определенных разрешений, по сути, заметая проблемы под ковер.

Разумеется, нет необходимости повторять, что **пороги для срабатывания медиазапросов должны диктоваться не конкретными устройствами**, а самим дизайном. И не только потому, что существует слишком много разнообразных устройств, на которых веб-сайт должен хорошо смотреться в любом возможном разрешении (особенно если учитывать также будущие разработки), но и потому, что на настольном компьютере веб-сайт может просматриваться в окне любого размера. Если вы уверены, что ваш дизайн хорошо работает на экране любого возможного размера, то какая вам разница, какое разрешение поддерживают конкретные устройства?

Следование принципам, описанным в разделе «Минимизируйте дублирование кода», также помогает в реализации гибкого дизайна, ведь вам не приходится переопределять лишние объявления в медиазапросах, что в конечном итоге минимизирует сопутствующие им издержки.

Вот еще несколько советов, как избежать ненужных медиазапросов:

- используйте процентные, а не фиксированные значения ширины. Когда это невозможно, используйте единицы длины, привязанные к окну просмотра (`vw`, `vh`, `vmin`, `vmax`), которые разрешаются в дробные части от ширины или длины окна просмотра;
- если вам требуется фиксированная ширина для большего разрешения, используйте `max-width`, а не `width`, для того чтобы дизайн мог адаптироваться к меньшему разрешению без помощи медиазапросов;
- не забывайте устанавливать значение `100%` для свойства `max-width` заменяемых элементов, таких как `img`, `object`, `video` и `iframe`;
- если фоновое изображение должно покрывать контейнер целиком, реализовать это можно с помощью `background-size: cover`, независимо от размера требуемого контейнера. Однако помните, что пропускная способность не бесконечна и не всегда разумно добавлять большие изображения, которые будут масштабироваться посредством CSS для дизайна, предназначенного для экранов мобильных устройств;
- выстраивая изображения (или другие элементы) в сетку из строк и столбцов, делайте так, чтобы количество столбцов диктовалось шириной окна просмотра. С этим вам могут помочь макет гибкого поля (также известный как Flexbox — Flexible Box Layout) и `display: inline-block` и обтекание обычным текстом;

- при использовании столбцов для отображения текста указывайте `column-width`, а не `column-count`, для того чтобы в маленьком разрешении весь текст выводился только в одном столбце.

В целом, идея заключается в том, чтобы стремиться к **текучим макетам и заданию размеров между точками прерывания медиазапросов с помощью относительных значений.** Когда дизайн достаточно гибок, сделать его отзывчивым можно с помощью всего лишь пары коротких медиазапросов. Дизайнеры из Basecamp говорили в точности об этом в конце 2010 года:

> *Выясняется, что создание макета, работающего на широком диапазоне устройств, — это, по сути, вопрос добавления пары медиазапросов CSS к готовому продукту. Ключ к тому, чтобы это оказалось для вас простейшей задачей, кроется в гибкости макета. Когда макет уже сам по себе текучий, оптимизировать его для небольших экранов означает всего лишь схлопнуть несколько полей для максимизации доступного пространства и скорректировать расположение боковых врезок для случаев, когда экран слишком узкий, чтобы на нем уместились два столбца.*
>
> **Experimenting with responsive design in Iterations**
> (http://signalvnoise.com/posts/2661-experimenting-with-responsive-design-in-iterations)

Если вы внезапно поняли, что вам требуется целая куча медиазапросов, чтобы адаптировать дизайн к экрану меньшего (или большего) размера, сделайте шаг назад и еще раз проанализируйте структуру своего кода, так как велика вероятность, что отзывчивость — не единственная ваша проблема.

Используйте сокращения с умом

Как вы, вероятно, знаете, следующие две строки CSS-кода не эквивалентны:

```
background: rebeccapurple;

background-color: rebeccapurple;
```

В первой строке используется сокращение, которое всегда обеспечивает фон, равномерно залитый цветом `rebeccapurple`. Элемент же, для которого использована полная запись (`background-color`), может в результате содержать розовый градиент, фотографию кошки или что угодно еще, поскольку на него может распространяться действие объявления `background-image`. Эта проблема чаще всего возникает именно при использовании полной записи: вы не сбрасываете значения всех остальных свойств, которые могут влиять на то, чего вы пытаетесь достичь.

Разумеется, можно попытаться **задать значения всех свойств в полной записи** и заявить, что дело сделано, но с высокой вероятностью какие-то вы все же забудете. Или же рабочая группа CSS представит в будущем больше вариантов полной записи, а в вашем коде не будет предусмотрено сбрасывания этих свойств. Не бойтесь сокращений. Это **хороший защитный стиль кодирования, подготавливающий ваш код к будущим изменениям**. Тем не менее **иногда вы можете намеренно использовать каскадные свойства для всего остального, как мы сделали с цветными вариантами кнопок в разделе «Минимизируйте дублирование кода»**.

Полные варианты записи также бывает удобно применять в комбинации с сокращениями, для того чтобы делать код более емким (в соответствии с принципом DRY) в свойствах, значения которых представляют собой списки с разделительной запятой, таких как свойства `background`. Лучше всего объяснить это на примере:

```
background: url(tr.png) no-repeat top right / 2em 2em,
            url(br.png) no-repeat bottom right / 2em 2em,
            url(bl.png) no-repeat bottom left / 2em 2em;
```

Обратите внимание, что значения `background-size` и `background-repeat` повторяются три раза, несмотря на то что для всех изображений они одинаковые. Мы можем воспользоваться преимуществом одного из правил разворачивания списков CSS, которое гласит, что **если указано только одно значение, то оно разворачивается и распространяется на все элементы в списке**. Это позволит нам переместить повторяющиеся значения в свойства с полной записью:

```
background: url(tr.png) top right,
            url(br.png) bottom right,
            url(bl.png) bottom left;
background-size: 2em 2em;
background-repeat: no-repeat;
```

Теперь значения `background-size` и `background-repeat` можно изменить с помощью всего одной правки вместо трех. Вы будете сталкиваться с применением этой техники на протяжении всей книги.

Нужно ли использовать препроцессор?

Вы наверняка слышали о препроцессорах CSS, таких как LESS (http://lesscss.org), Sass (http://sass-lang.com) и Stylus (http://stylus-lang.com/). Они предлагают несколько удобных инструментов для разработки CSS, таких как переменные, примеси, функции, вложенные правила, манипулирование цветом и многие другие.

При правильном использовании они помогают делать код более гибким в крупных проектах, когда возможностей одного только CSS становится недостаточно. Мы всегда стремимся создавать надежный, гибкий и емкий CSS-код, но иногда ограничения самого языка оказываются непреодолимыми. С другой стороны, сами препроцессоры также способны создавать определенные сложности:

- вы теряете контроль над размерами файлов и сложностью вашего CSS-кода. Емкий, краткий код после компиляции может превратиться в исполинское чудище, которое вам придется пересылать по проводам;
- **задача поиска и устранения ошибок становится сложнее**, так как CSS-код, который вы видите в инструментах разработки, — это не тот CSS-код, который вы пишете. Эту проблему несколько смягчает появление новых инструментов отладки в *SourceMaps*. SourceMaps — это новая классная технология, назначение которой — устранять подобные сложности. Это достигается за счет того, что она с точностью до номера строки сообщает браузеру, какой CSS-код препроцессора какому сгенерированному CSS-коду соответствует;

ЗАНИМАТЕЛЬНАЯ СТРАНИЧКА.
СТРАННЫЙ СОКРАЩЕННЫЙ СИНТАКСИС

Возможно, вы заметили, что в предыдущем примере в сокращенном варианте `background` после определения `background-size` необходимо также указывать значение `background-position` (даже если оно не отличается от первоначального) и отделять его слешем (`/`).

Это практически всегда делается в целях разрешения противоречий. Определенно, в примере выше очевидно, что `top right` — это `background-position`, а `2em 2em` — `background-size`, независимо от того, в каком порядке они указаны. Однако представьте, что мы используем значения вроде `50% 50%`. Что это — `background-size` или `background-position`? Когда вы используете полную запись, парсер CSS сразу понимает, что имеется в виду. Однако в сокращении, где нет никаких имен свойств, парсеру необходима подсказка, для того чтобы понять, к чему относится это `50% 50%`. Вот для чего служит слеш.

С большинством сокращений подобных проблем с необходимостью устранения неоднозначности не возникает, и для них значения свойств можно перечислять в любом порядке. Однако всегда сверяться с точным синтаксисом во избежание неприятных сюрпризов — это хорошая практика кодирования. Если вы знакомы с регулярными выражениями и грамматиками, то можете также проверить грамматику для свойства в соответствующей спецификации; это, вероятно, самый быстрый способ узнать, определен ли какой-то конкретный порядок.

- из-за препроцессоров в процессе разработки могут возникать определенные **задержки**. Хотя в целом эти инструменты работают довольно быстро, компиляция кода в CSS все же требует какого-то времени, и вам приходится выжидать пару секунд, прежде чем вы сможете проверить результат;
- каждая новая абстракция порождает дополнительные усилия, которые кому-то придется затратить, чтобы начать работу с нашим кодом. Нам нужно либо сотрудничать только с хорошо знакомыми с диалектом людьми, либо обучать их этому диалекту. Таким образом, мы **либо ограничены в выборе соавторов, либо обязаны потратить дополнительное время на обучение**; оба варианта малопривлекательны;
- и давайте не забывать о *законе протекающих абстракций:* «Все нетривиальные абстракции в той или иной степени допускают утечку». Препроцессоры пишутся людьми, и, как и любая нетривиальная программа, когда-либо написанная человеческими существами, **они содержат собственные ошибки**, могущие быть весьма коварными, ведь мы обычно даже не подозреваем, что причина проблем с нашим CSS-кодом может скрываться в коде препроцессора.

Помимо проблем, перечисленных выше, существует также риск привыкания к препроцессорам, когда разработчики продолжают использовать их, даже если необходимость в этом отсутствует, — в небольших проектах или в будущем, после того как их любимые возможности уже добавляются в чистый CSS. Удивлены? Да, **многие возможности, вдохновленные препроцессорами, в итоге оказались частью чистого CSS:**

- уже существует черновик, озаглавленный **CSS Custom Properties for Cascading Variables** и описывающий пользовательские свойства с функциональностью переменных (http://w3.org/TR/css-variables-1);
- функция `calc()` из **CSS Values & Units Level 3** — это не просто мощный инструмент выполнения вычислений; уже сегодня реализована ее широкая поддержка;
- функция `color()` из **CSS Color Level 4** (http://dev.w3.org/csswg/csscolor) дает возможность манипулировать цветом;
- в рамках рабочей группы CSS ведутся серьезные обсуждения вложенности, и в прошлом даже существовала посвященная этому черновая спецификация.

Обратите внимание, что «родные» возможности, подобные этим, обычно **намного мощнее заменителей, предоставляемых препроцессорами**, благодаря своей динамичности. Например, препроцессор не имеет ни малейшего представления о том, как выполнять вычисления вроде `100% - 50px`, так как значение, в которое должны разрешиться эти проценты, неизвестно до тех пор, пока страница не будет фактически визуализирована. Однако у функции `calc()` из чистого CSS

нет никаких сложностей с оценкой подобных выражений. Схожим образом подобное использование переменных невозможно в случае, если это переменные, предоставляемые препроцессором:

```
ul { --accent-color: purple; }
ol { --accent-color: rebeccapurple; }
li { background: var(--accent-color); }
```

Вы понимаете, что мы сделали? В качестве фона элементов списка в упорядоченных списках будет использоваться цвет `rebeccapurple`, тогда как в неупорядоченных списках элементы будут отображаться на фоне цвета `purple`. Попробуйте добиться этого с препроцессором! Разумеется, в данном случае мы могли бы всего лишь использовать селекторы-потомки, но суть примера в том, чтобы продемонстрировать динамичность этих переменных.



Рис. 1.13. Myth (http://myth.io) — это экспериментальный препроцессор, который, вместо того чтобы использовать собственный синтаксис, эмулирует «родные» возможности CSS, по сути, играя роль полифилла CSS

Так как большинство из упомянутых выше возможностей чистого CSS сегодня поддерживаются еще недостаточно хорошо, очень часто, если сопровождаемость кода важна для вас (а это должно быть так), использования препроцессоров не избежать. Я советую начинать каждый проект с чистого CSS и прибегать

к помощи препроцессора только в том случае, если при реализации необходимой функциональности становится невозможно соблюдать принципы DRY. Для того чтобы не впасть в полную зависимость от препроцессоров и не использовать их в тех ситуациях, когда они в действительности не нужны, **подключение препроцессора к проекту всегда должно быть осознанным решением**, а не бездумным первым шагом, который по умолчанию делается в каждом новом проекте.

Не забывайте, что для манипулирования этими и другими «родными» возможностями CSS можно также использовать сценарии. Например, поменять значение переменной можно в коде JS.

2

Фон и рамки

1 Полупрозрачные рамки

ПРЕДВАРИТЕЛЬНЫЕ ТРЕБОВАНИЯ

Знание цветов RGBS/HSLA

Проблема

Вам, вероятно, достаточно часто приходилось возиться с инструментами создания полупрозрачных цветов в CSS, такими как `rgba()` и `hsla()`. В 2009 году, когда мы, наконец, получили возможность использовать их в своем дизайне, они произвели настоящий фурор — и это несмотря на необходимость продумывать пути отхода, разнообразные «костыли» и даже уродливые поделки с фильтрами в IE для самых отважных. Однако их применение в промышленных условиях ограничивалось в основном фонами. Существует три главные причины, почему так происходило:

- часть первых экспериментаторов не поняла, что эти новые цветовые форматы в действительности были обычными цветами, такими как `#ff0066` и `orange`, и работала с ними как с изображениями, используя только для фонов;
- обходные пути было намного проще реализовать для фонов, чем для любых других свойств. Например, альтернативным решением для полупрозрачного фона могло быть простое полупрозрачное изображение площадью в один пиксел. Для других свойств единственной альтернативой оставался сплошной цвет;
- применение их для других свойств, таких как рамки, зачастую оказывалось делом далеко не простым, и скоро мы узнаем почему.

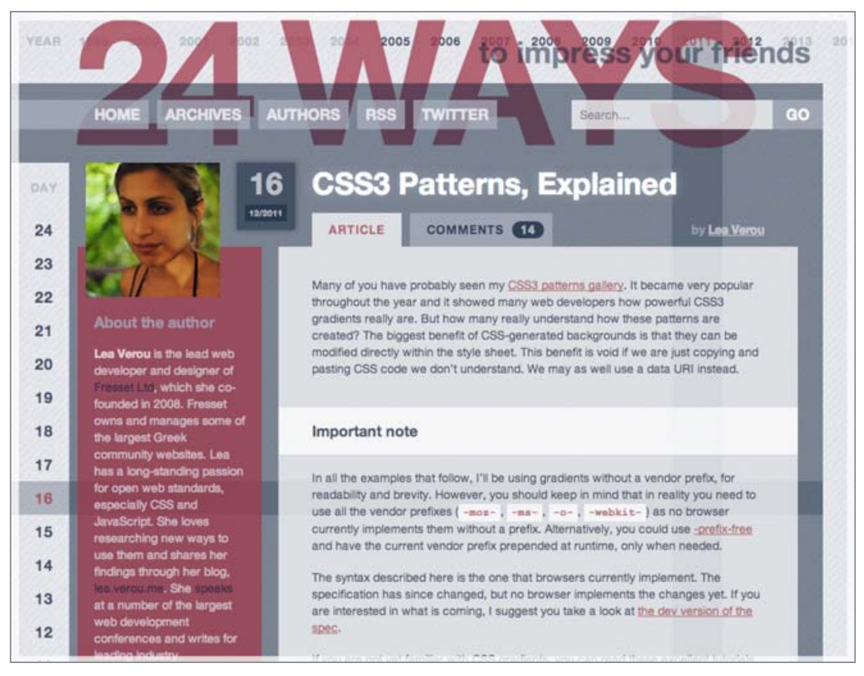


Рис. 2.1. 24ways.org был одним из первых веб-сайтов, в дизайне которых еще в 2008 году использовались настоящие полупрозрачные цвета, хоть и для фонов (автор дизайна — Тим Ван Дамм)

Предположим, что мы хотим для оформления контейнера использовать белый фон и полупрозрачную белую рамку, сквозь которую будет просвечивать основной фон. Наша первая попытка, вероятно, будет выглядеть примерно так:

```
border: 10px solid hsla(0,0%,100%,.5);
background: white;
```

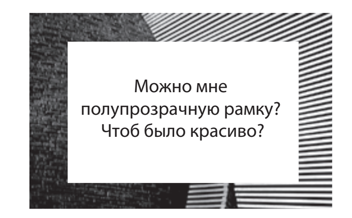


Рис. 2.2. Наша первая попытка добиться эффекта полупрозрачной рамки

Если у вас нет твердого понимания, как работают фоны и рамки, результат (показанный на рис. 2.2) может шокировать. Куда делась рамка? И если невозможно достичь эффекта полупрозрачной рамки, используя полупрозрачный цвет для рамки, то как это вообще сделать?

Решение

Хотя это и не очевидно, наша рамка на месте и никуда не делась. По умолчанию фон растягивается, заполняя в том числе область рамки, что можно легко проверить, применив старый добрый эффект пунктирной рамки к элементу, для



Рис. 2.3. По умолчанию фон распространяется также и на область рамки

которого определен фон (рис. 2.3). Это не играет особой роли, когда вы используете сплошные непрозрачные рамки, но в нашем примере способно полностью изменить дизайн. Вместо полупрозрачной белой рамки, сквозь которую просвечивает прелестная фоновая картинка, мы получили полупрозрачную белую рамку на непрозрачном белом фоне, которую не отличить от обычной белой рамки.

В CSS 2.1 фон именно так и работал. Нам нужно было просто смириться и продолжать жить с этим. К счастью, с появлением стандарта **Backgrounds & Borders Level 3** (http://w3.org/TR/css3-background) мы обрели возможность корректировать поведение фона в соответствии с насущными требованиями дизайна, применяя для этого свойство `background-clip`. Его первоначальное значение равно `border-box`, что означает, что фон обрезается по краю *ограничивающей рамки*. Если мы хотим, чтобы наш фон не растягивался под рамку, то для этого всего лишь нужно присвоить свойству значение `padding-box`, которое приказывает браузеру обрезать фон по краю забивки:



Рис. 2.4. Проблема решена с помощью свойства `background-clip`

```
border: 10px solid hsla(0,0%,100%,.5);
background: white;
background-clip: padding-box;
```

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**translucent-borders**

СВЯЗАННЫЕ СПЕЦИФИКАЦИИ

CSS Backgrounds & Borders: http://w3.org/TR/css-backgrounds

2 Несколько рамок

ПРЕДВАРИТЕЛЬНЫЕ ТРЕБОВАНИЯ

Простейшие варианты использования `box-shadow`

Проблема

В стародавние времена, когда спецификация **Backgrounds & Borders Level 3** (http://w3.org/TR/css3-background) была еще на стадии черновика, в рабочей группе CSS велось активное обсуждение вопроса, нужно ли разрешать использование сразу нескольких рамок — нескольких фоновых изображений. К сожалению, тогда все сошлись на том, что вариантов использования нескольких рамок слишком мало и разработчики всегда могут прибегнуть к помощи `border-image` для достижения того же эффекта. Однако рабочая группа упустила из виду важность наличия возможности гибко настраивать рамки в CSS-коде, из-за чего разработчикам для имитации нескольких рамок пришлось прибегать к корявым трюкам вроде наложения друг на друга множества элементов. И все же существуют более удачные способы решения этой задачи, не приводящие к загрязнению кода бесполезными лишними элементами.

Решение с использованием box-shadow

К настоящему моменту большинство из нас уже имеет (слишком большой) опыт использования `box-shadow` для создания теней. Однако мало кто знает, что это свойство принимает четвертый параметр (называемый **радиусом размазывания**, *spread radius*), который **делает тень больше** (положительные значения) или **меньше** (отрицательные значения) на указанную величину. Положительный



Рис. 2.5. Имитация контура с помощью свойства `box-shadow`

радиус размазывания в сочетании с нулевым смещением и нулевым размыванием создает «тень», больше похожую на сплошную рамку (рис. 2.5):

```
background: yellowgreen;
box-shadow: 0 0 0 10px #655;
```

Не слишком впечатляет, ведь точно такую же рамку можно создать с использованием свойства `border`. Но хорошая новость в том, что благодаря свойству `box-shadow` **мы можем создать сколько угодно рамок, просто разделив наборы значений запятой**. То есть добавить к предыдущему примеру вторую рамку ярко-розового цвета `deeppink` совсем несложно:



Рис. 2.6. Имитация двух контуров с помощью свойства box-shadow

```
background: yellowgreen;
box-shadow: 0 0 0 10px #655, 0 0 0 15px
deeppink;
```

Единственное, о чем необходимо помнить, — что рамки, создаваемые свойством `box-shadow`, накладываются друг на друга, причем наверху оказывается та, которая в строке значений указана первой. Например, в предыдущем коде нам нужна была внешняя рамка шириной `5px`, поэтому мы указали радиус размазывания `15px` (`10px` + `5px`). При необходимости после «контуров» можно добавить еще один набор значений для обычной тени:

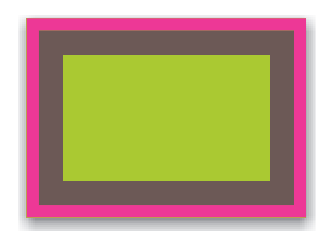


Рис. 2.7. Добавление настоящей тени после «контуров»

```
background: yellowgreen;
box-shadow: 0 0 0 10px #655,
            0 0 0 15px deeppink,
            0 2px 5px 15px rgba(0,0,0,.6);
```

Решение с тенью хорошо работает в большинстве случаев, но существует несколько тонкостей, о которых не стоит забывать:

- тени работают *не совсем* так, как рамки: они не влияют на макет и на них не распространяется действие свойства `box-sizing`. Однако имитировать дополнительное пространство, которое заняла бы рамка, можно с помощью забивки или полей (в зависимости от того, определена тень как `inset` или нет);

- продемонстрированный выше метод позволяет создавать фальшивые «рамки» **снаружи** элементов. Подобные рамки не умеют перехватывать события мыши, такие как наведение или щелчок. Если это важно для вас, то можно добавлять ключевое слово `inset`, чтобы тени рисовались внутри элемента. Помните только, что вам придется добавить больше забивки, чтобы обеспечить необходимое пространство.

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**multiple-borders**

Решение с использованием outline

В некоторых случаях, **если нам требуются только две рамки**, можно определить обычную рамку, а эффект внешнего контура создать с помощью свойства `outline`. Это обеспечивает гибкость в выборе стиля рамки (а вдруг мы захотим, чтобы вторая рамка была пунктирной?), тогда как метод с `box-shadow` позволяет создавать только сплошные рамки. Вот как в данном случае будет выглядеть код для создания эффекта, показанного на рис. 2.6:

```
background: yellowgreen;
border: 10px solid #655;
outline: 15px solid deeppink;
```

Что еще хорошо в контурах, так это то, что расстояние от границ элемента можно менять путем определения свойства `outline-offset`, которое способно принимать даже отрицательные значения. Это может быть полезно для создания целого ряда интересных эффектов. Например, на рис. 2.8 вы видите простой пример эффекта прошитого края.

Однако у данного метода есть несколько ограничений:

- как уже упоминалось выше, он подходит для определения только двух «рамок», так как свойство `outline` не принимает список значений, разделенных запятой. Если вам требуется больше, то единственный вариант — прибегнуть к технике, описанной в предыдущем разделе;
- контуры не обязаны подчиняться свойству `border-radius`, поэтому даже если углы рамки скруглены, контур может оказаться прямоугольным



Рис. 2.8. Используйте отрицательное значение `outline-offset` с пунктирной (`dashed`) рамкой, чтобы создать простейший эффект прошитого края



Рис. 2.9. Контуры, создаваемые с помощью свойства `outline`, не повторяют скругленные очертания элемента, но в будущем это может измениться

(рис. 2.9). Обратите внимание, что рабочая группа CSS считает такое поведение ошибкой, и в будущем, скорее всего, поведение контуров будет изменено, для того чтобы действие `border-radius` распространялось и на них тоже;

- согласно спецификации **CSS User Interface Level 3** (http://w3.org/TR/css3-ui), «контуры могут быть не прямоугольными». Хотя в большинстве случаев они получаются прямоугольными, если вы собираетесь пользоваться этим методом, мысленно отметьте, что результат следует тщательно протестировать в разных браузерах.

СВЯЗАННЫЕ СПЕЦИФИКАЦИИ

CSS Backgrounds & Borders: http://w3.org/TR/css-backgrounds

CSS Basic User Interface: http://w3.org/TR/css3-ui

3 Гибкое позиционирование фона

Проблема

Довольно часто у нас возникает необходимость позиционировать фоновое изображение со сдвигом относительно не верхнего левого угла, а какого-либо другого, например правого нижнего. В CSS 2.1 можно было либо задать сдвиг относительно верхнего левого угла, либо использовать ключевое слово для одного из остальных трех. Однако чаще всего мы все же хотим, чтобы между фоновым изображением и ближайшим углом оставалось немного пустого места (что-то вроде забивки), чтобы избежать эффекта, показанного на рис. 2.10.



Рис. 2.10. `background-position: bottom right;` обычно не обеспечивает приятного глазу результата, так как изображение вплотную прижимается к сторонам контейнера

Для контейнеров фиксированного размера это возможно и средствами CSS 2.1, хотя решение получается не из простых: нужно подсчитать, каким должен быть сдвиг фонового изображения относительно верхнего левого угла, отталкиваясь от фиксированной длины и ширины контейнера и желаемого сдвига относительно правого нижнего угла. Однако для элементов переменного размера (вследствие того, что их содержимое может меняться) это невозможно. В итоге разработчикам приходилось задавать приблизительные значения, например указывать местоположение фона процентным значением немного меньше 100%, скажем 95%. Определенно, в современном CSS должен найтись способ получше!

Решение с использованием расширенного свойства background-position

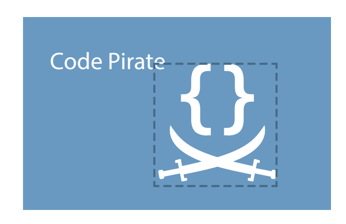


Рис. 2.11. Задание сдвига относительно разных сторон контейнера; фоновое изображение здесь обведено пунктирным контуром, чтобы было понятнее, как работает сдвиг

В **CSS Backgrounds & Borders Level 3** (http://w3.org/TR/css3-background) свойство `background-position` было обновлено и теперь поддерживает задание **сдвига относительно любого угла** — для этого **перед значениями сдвига необходимо указывать определенные ключевые слова.** Например, если мы хотим, чтобы наше фоновое изображение было сдвинуто на `20px` от правого края контейнера и на `10px` от нижнего края, то мы можем использовать следующий код:

```
background: url(code-pirate.svg) no-repeat #58a;
background-position: right 20px bottom 10px;
```

Результат вы можете видеть на рис. 2.11. Последний шаг — обеспечить достойный путь отступления. Предыдущая версия кода приведет к тому, что в браузерах, которые не поддерживают расширенный синтаксис `background-position`, фоновое изображение прилипнет к верхнему левому краю (позиция по умолчанию), что выглядит просто ужасно, не говоря уже о том, что текст при этом прочитать будет невозможно (рис. 2.12). Хорошим обходным путем будет добавление старой доброй позиции `bottom right` в сокращение `background`:

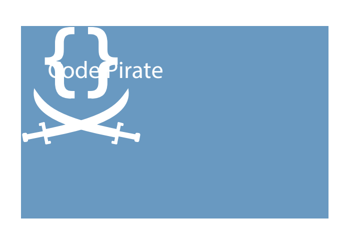


Рис. 2.12. Необходимо указать обходной путь, чтобы пользователи старых браузеров не натолкнулись на подобную картину

```
background: url(code-pirate.svg)
            no-repeat bottom right #58a;
background-position: right 20px bottom 10px;
```

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**extended-bg-position**

Решение с использованием background-origin

Один из самых распространенных случаев, когда требуется задать сдвиг относительно угла контейнера, — необходимость совместить фоновое изображение с забивкой. Решение с использованием расширенного свойства `background-position`, которое мы рассмотрели выше, выглядело бы следующим образом:

```
padding: 10px;
background: url(code-pirate.svg) no-repeat #58a;
background-position: right 10px bottom 10px;
```

Результат показан на рис. 2.13. Как вы видите, решение работает, но оно не соответствует принципам DRY: каждый раз при изменении ширины забивки нам необходимо обновлять это значение в трех разных местах! К счастью, существует более простой способ добиться нужного эффекта, который к тому же автоматически учитывает значение забивки, без необходимости заново переопределять сдвиги.



Рис. 2.13. Смещение фонового изображения на расстояние, равное значению забивки

Скорее всего, за свою карьеру веб-разработчика вы немало раз упоминали в коде такие вещи, как `background-position: top left;`. Но задавались ли вы когда-нибудь вопросом, *какой именно верхний левый угол имеется в виду под top left*? Как вы, вероятно, знаете, каждый элемент состоит из трех полей (рис. 2.14): поля рамки, поля забивки и поля содержимого. На верхний левый угол какого из этих полей ориентируется свойство `background-position`?

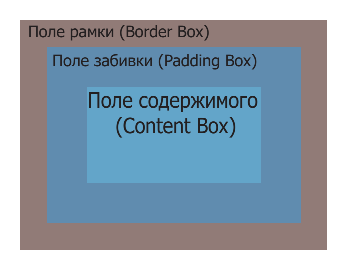


Рис. 2.14. Модель полей

По умолчанию `background-position` ссылается на поле забивки; это сделано для того, чтобы рамки не закрывали собой фоновые изображения. Следовательно, `top left` — это по умолчанию верхний левый внешний угол поля забивки. В **Backgrounds & Borders Level 3** (http://w3.org/TR/css3-background), однако, у нас появилось новое свойство, позволяющее изменить данное поведение: `background-origin`. Его значение по умолчанию (вполне предсказуемо) равно `padding-box`. Если изменить его на `content-box`, как в следующем фрагменте кода, то ключевые слова для обозначения стороны и угла, которые мы используем с `background-position`, будут ссылаться на край поля содержимого (по сути, это означает, что любые фоновые изображения будут сдвигаться относительно сторон/углов на величину забивки):

```
padding: 10px;
background: url("code-pirate.svg") no-repeat #58a
            bottom right; /* или 100% 100% */
background-origin: content-box;
```

Визуально результат выглядит точно так же, как на рис. 2.13, но код лучше соответствует принципам DRY. Помните также, что при необходимости эти две



Не забывайте добавлять пробелы вокруг всех операций - и + внутри функции calc(), иначе синтаксический разбор вернет ошибку. Причина существования этого странного правила кроется в необходимости обеспечить совместимость с будущими разработками: в будущем внутри calc(), возможно, будут разрешены ключевые слова, а они могут содержать дефисы.

техники можно совмещать! Если вам нужно, чтобы смещение в целом соответствовало величине забивки, но фоновое изображение все же было слегка сдвинуто внутрь или наружу, то можете использовать background-origin: content-box совместно с дополнительными значениями смещения, определенными посредством background-position.

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**background-origin**

Решение с использованием calc()

Давайте вспомним исходную формулировку задачи: мы хотим поместить наше фоновое изображение на расстоянии 10px от нижнего края и 20px от правого края. Однако если мыслить в терминах смещения относительно верхнего левого угла, то нам требуется смещение на 100% - 20px по горизонтали и на 100% - 10px по вертикали. К счастью, функция calc() позволяет выполнять подобные вычисления и прекрасно поддерживает background-position:

```
background: url("code-pirate.svg") no-repeat;
background-position: calc(100% - 20px) calc(100% - 10px);
```

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**background-position-calc**

СВЯЗАННЫЕ СПЕЦИФИКАЦИИ

CSS Backgrounds & Borders: http://w3.org/TR/css-backgrounds

CSS Values & Units: http://w3.org/TR/css-values

4 Внутреннее скругление

ПРЕДВАРИТЕЛЬНЫЕ ТРЕБОВАНИЯ

`box-shadow`, `outline`, секрет «Несколько рамок»

Проблема

Иногда нам требуется контейнер, скругленный только изнутри, внешние углы рамки/контура которого все так же остаются прямыми, как на рис. 2.15. Это интересный неизбитый эффект. Подобного эффекта легко добиться с помощью двух элементов:

У меня замечательные скругленные углы внутри, разве я не прекрасен?

Рис. 2.15. Контейнер с контуром и скруглением только изнутри

HTML

```
<div class="something-meaningful"><div>
    I have a nice subtle inner rounding,
    don't I look pretty?
</div></div>

.something-meaningful {
    background: #655;
    padding: .8em;
}
.something-meaningful > div {
    background: tan;
    border-radius: .8em;
    padding: 1em;
}
```

Я грустный элемент, потому что мой контур не повторяет мои скругленные углы :-(

Рис. 2.16. Использование свойства `outline` на скругленном элементе

Я счастливый элемент, потому что мой фальшивый контур повторяет мои скругленные углы :-)

Рис. 2.17. Использование свойства `box-shadow` без смещения и размытия на элементе со скругленными углами

Я замечательный внутренний контур, разве я не прекрасен?

Рис. 2.18. Здесь контур подсвечен черным цветом, а тень — пурпурным; это помогает понять, что происходит с этим элементом. Обратите внимание, что контур рисуется поверх тени

Это прекрасно работает, но нам приходится использовать два элемента, тогда как в действительности требуется только один. Можно ли достичь того же эффекта с одним элементом?

Решение

Предыдущее решение более гибкое — оно позволяет пользоваться всеми преимуществами фонов. Например, если мы хотим, чтобы наша «рамка» была закрашена не простым однородным цветом, а обладала какой-то текстурой, добиться этого довольно просто. Однако если мы имеем дело исключительно со старыми добрыми сплошными цветами, то нам хватит и одного элемента (хотя это и грязный способ). Взгляните на следующий фрагмент CSS-кода:

```
background: tan;
border-radius: .8em;
padding: 1em;
box-shadow: 0 0 0 .6em #655;
outline: .6em solid #655;
```

Можете угадать, каким будет визуальный результат? Данный код порождает эффект, показанный на рис. 2.15. По сути, мы воспользовались тем фактом, что контуры не повторяют скругление углов элемента (и, следовательно, обладают прямыми углами, как показано на рис. 2.16), а тени (`box-shadow`), наоборот, скругляются (рис. 2.17). Следовательно, если мы наложим их друг на друга, то `box-shadow` закроет «дыры», которые контур оставляет по углам (рис. 2.18), и комбинация этих свойств даст нам необходимый эффект. На рис. 2.18 тень и контуры подсвечены разными цветами в качестве визуального пояснения.

Обратите внимание, что в действительности `box-shadow` не требуется размазывать на величину контура — хватит размазывания, достаточного для заполнения этих «дырок». В действительности, если размазывание будет равно ширине контура, то в некоторых браузерах это может привести к появлению визуальных артефактов, поэтому я рекомендую использовать значение чуть меньше. Это сразу вызывает вопрос: **какова минимальная величина размазывания, которую необходимо указать, чтобы закрыть «дыры»?**

Для того чтобы ответить на этот вопрос, необходимо вспомнить теорему Пифагора, которую мы все изучали в школе и которая позволяет вычислять длины сторон прямоугольных треугольников. Согласно теореме, длина гипотенузы (самой длинной, диагональной стороны треугольника) равна $\sqrt{a^2+b^2}$, где a и b — длины катетов. Если катеты равны по величине, то формула превращается в $\sqrt{2a^2}=a\sqrt{2}$.

Возможно, вы задаетесь вопросом, какое отношение геометрия уровня средней школы имеет к эффекту скругления внутреннего угла. Взгляните на рис. 2.19: эта схема дает визуальную подсказку относительно того, как вычислить минимальную ширину размазывания. В нашем случае значение `border-radius` равно $.8em$, следовательно, минимальное размазывание равно $.8(\sqrt{2}-1)\approx.33137085em$. Все, что нам остается, — слегка округлить это значение вверх и указать радиус размазывания $.34em$. Чтобы избежать необходимости проводить подобные вычисления каждый раз, когда вам требуется данный эффект, можно попросту использовать половину радиуса угла, что гарантированно даст достаточно большое значение, так как $\sqrt{2}-1<0{,}5$.

Обратите внимание также, что данные вычисления обнаруживают **еще одно ограничение этого метода**: для того чтобы наш эффект сработал, радиус размазывания должен быть меньше ширины контура, но больше $(\sqrt{2}-1)r$, где r — это значение `border-radius`. Это означает, что если ширина контура меньше $(\sqrt{2}-1)r$, то применить данный эффект невозможно.



Почему этот способ **грязный**? Потому что **он полагается на тот факт, что контуры не повторяют скругление углов**, однако нет никакой гарантии, что это поведение не изменится. В настоящее время спецификация дает разработчикам браузеров право самостоятельно принимать решения относительно того, как должны рисоваться контуры, но в будущем **планируется выпустить явную рекомендацию следовать скруглению — данное решение уже принято рабочей группой CSS.** Когда же оно будет фактически реализовано в браузерах, пока что остается загадкой.

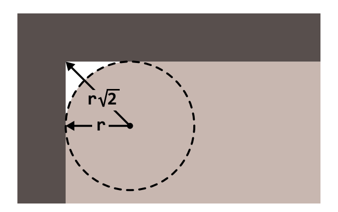


Рис. 2.19. Если радиус рамки равен r, то длина от центра окружности `border-radius` до угла контурного прямоугольника равна $r\sqrt{2}$, что означает, что минимально возможное размазывание равно $r\sqrt{2}-r=(\sqrt{2}-1)r$

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**inner-rounding**

СВЯЗАННЫЕ СПЕЦИФИКАЦИИ

CSS Backgrounds & Borders: http://w3.org/TR/css-backgrounds

CSS Basic User Interface: http://w3.org/TR/css3-ui

5 Фон в полоску

ПРЕДВАРИТЕЛЬНЫЕ ТРЕБОВАНИЯ

Знание линейных градиентов CSS, свойства `background-size`

Проблема

Полоски всех размеров, цветов и углов так же, если не больше, распространены в веб-дизайне, как и в любых других видах визуального дизайна, от журналов до обоев. Однако процесс реализации такого дизайна далек от идеала. Обычно мы создаем отдельное растровое изображение и каждый раз, когда возникает необходимость внести изменения, прибегаем к помощи графического редактора. Некоторые вместо этого используют файлы SVG, но это особый формат, синтаксис которого далек от дружественного. Правда, было бы здорово, если бы мы могли создавать полоски прямо в CSS? Вас это может удивить, но это возможно!

Решение

Предположим, что у нас есть простейший вертикальный линейный градиент, от `#fb3` до `#58a` (рис. 2.20):

```
background: linear-gradient(#fb3, #58a);
```

Теперь попробуем немного приблизить друг к другу границы перехода цвета (рис. 2.21):

```
background: linear-gradient(#fb3 20%, #58a 80%);
```

Теперь верхние 20% нашего контейнера заполнены сплошным цветом `#fb3`, а нижние 20% — сплошным цветом `#58a`. Настоящий градиент занимает только 60% высоты контейнера. Если мы еще сильнее сдвинем границы перехода цвета (40% и 60% соответственно, как показано на рис. 2.22), то высота градиента станет еще меньше. Возникает вопрос: а что произойдет, если границы перехода цвета встретятся на одном уровне?

```
background: linear-gradient(#fb3 50%, #58a 50%);
```

> *Если в одной и той же позиции определено несколько границ перехода цвета, то они образуют бесконечно малый переход от цвета, указанного в правиле первым, к цвету, указанному последним. Фактически в этой позиции вместо гладкого перетекания происходит просто резкая смена цвета.*
>
> CSS Image Values Level 3
> (http://w3.org/TR/css3-images)

Как вы видите на рис. 2.23, никакого градиента больше нет, только два участка сплошного цвета, каждый из которых занимает по половине нашего `background-image`. По сути, мы создали две большие горизонтальные полосы.

Так как градиенты — это всего лишь сгенерированные фоновые изображения, то с ними можно обходиться так же, как с любыми другими фоновыми изображениями, например, корректировать их размер с помощью `background-size`:

```
background: linear-gradient(#fb3 50%, #58a 50%);
background-size: 100% 30px;
```

Как видно на рис. 2.24, мы уменьшили высоту наших двух полосок до `15px` каждая. Так как наш фон повторяется, теперь весь контейнер заполнен горизонтальными полосами (рис. 2.25).



Рис. 2.20. Наша отправная точка

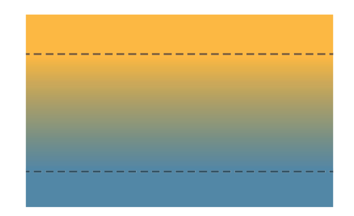


Рис. 2.21. Теперь градиент занимает 60% общей высоты элемента, а оставшаяся часть заполнена сплошными цветами; границы перехода цвета показаны пунктирными линиями

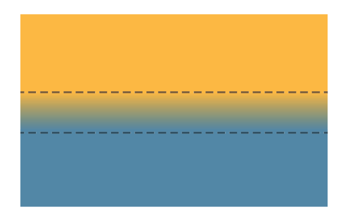


Рис. 2.22. Теперь градиент занимает только 20% общей высоты элемента, а оставшиеся части заполнены сплошными цветами; границы перехода цвета показаны пунктирными линиями



Рис. 2.23. Обе границы перехода цвета сейчас находятся на отметке 50%

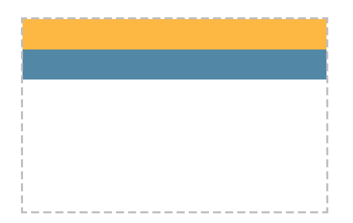


Рис. 2.24. Наш сгенерированный фон без повторения

Схожим образом можно создавать полосы неравной ширины, просто корректируя позиции границ перехода цвета (рис. 2.26):

```
background: linear-gradient(#fb3 30%, #58a 30%);
background-size: 100% 30px;
```

Для того чтобы избежать необходимости корректировать два значения каждый раз, когда нам нужно изменить ширину полосок, воспользуемся преимуществом, которое дает нам спецификация:

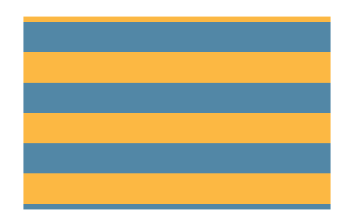


Рис. 2.25. Горизонтальные полосы — итоговый результат

> Если для границы перехода цвета задана позиция, меньшая чем позиция любой другой границы перехода цвета перед ней в списке, то следует установить ее позицию равной наибольшей позиции среди всех предшествующих границ перехода цвета.
>
> **CSS Images Level 3** (http://w3.org/TR/css3-images)

Это означает, что если для второго цвета мы зададим позицию на уровне `0`, то браузер скорректирует ее, увеличив до позиции предыдущей границы перехода цвета, — как раз то, что нам требуется. Следовательно, следующий фрагмент кода создает точно такой же градиент, который мы уже видели на рис. 2.26, и при этом лучше соответствует принципам DRY:



Рис. 2.26. Полосы неравной ширины

```
background: linear-gradient(#fb3 30%, #58a 0);
background-size: 100% 30px;
```

Создавать полоски с большим количеством цветов ничуть не сложнее. Например, следующий фрагмент кода определяет горизонтальные полосы трех разных цветов (рис. 2.27):

```
background: linear-gradient(#fb3 33.3%,
            #58a 0, #58a 66.6%, yellowgreen 0);
background-size: 100% 45px;
```

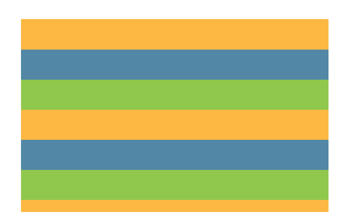


Рис. 2.27. Полосы трех цветов

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**horizontal-stripes**

Вертикальные полосы

Горизонтальные полосы реализовать проще всего, но на веб-сайтах, которые нам попадаются в Сети, встречаются фоны не только с горизонтальными полосами. Не менее распространены вертикальные полосы (рис. 2.28), а самые популярные и визуально интересные — это, вероятно, разные варианты диагональных полос. К счастью, градиенты CSS помогают справиться и с этими задачами, предлагая решения разной степени сложности.

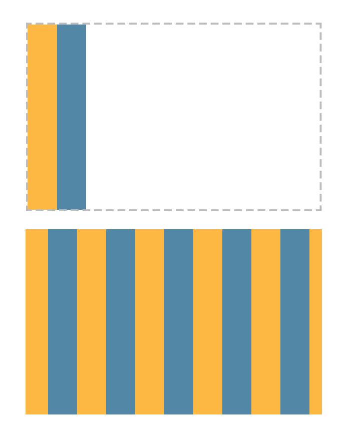


Рис. 2.28. Наши вертикальные полосы. *Вверху*: заполнение вертикальными полосами без повторения. *Внизу*: повторяющиеся полосы

Код, создающий вертикальные полосы, очень похож на предыдущий. Главное отличие — дополнительный первый аргумент, указывающий направление градиента. Мы могли бы указать его и при определении горизонтальных полос, но тогда нам было бы достаточно значения по умолчанию (`to bottom`). Помимо этого, в данном случае нам, очевидно, необходимо указать другие значения `background-size`:

```
background: linear-gradient(to right, /* или 90deg */
            #fb3 50%, #58a 0);
background-size: 30px 100%;
```

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**vertical-stripes**

Диагональные полосы

После горизонтальных и вертикальных полос кажется логичным попытаться создать диагональные полосы (с углом наклона 45°), еще раз изменив значение `background-size` и направление градиента, например, так:

```
background: linear-gradient(45deg,
            #fb3 50%, #58a 0);
background-size: 30px 30px;
```

Однако, как вы видите на рис. 2.29, этот способ не работает. Причина в том, что мы всего лишь повернули градиент *внутри* каждой плитки на 45°, а не определили повторяющуюся фоновую картинку. Вспомните растровые изображения,

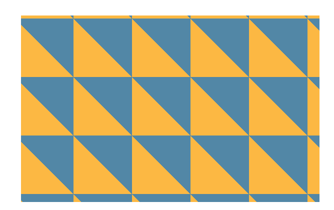


Рис. 2.29. Наша первая провальная попытка создать диагональные полосы



Рис. 2.30. Такие изображения стыкуются бесшовно, создавая аккуратные диагональные полосы; не кажется ли вам этот рисунок знакомым?

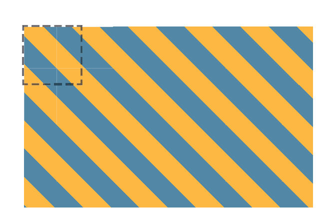


Рис. 2.31. Наши полосы под углом 45°; пунктирными линиями обозначена повторяющаяся плитка рисунка

которые мы обычно используем для создания диагональных полос, такие как показанные на рис. 2.30. Они включают четыре полосы, а не две, поэтому стыки между плитками совершенно не заметны. Именно такой рисунок нам необходимо создать с помощью CSS-кода, поэтому добавим еще парочку границ перехода цвета:

```
background: linear-gradient(45deg,
            #fb3 25%, #58a 0, #58a 50%,
            #fb3 0, #fb3 75%, #58a 0);
background-size: 30px 30px;
```

Результат показан на рис. 2.31. Как вы видите, нам удалось создать диагональные полосы, но они кажутся тоньше, чем горизонтальные и вертикальные из предыдущих примеров. Для того чтобы понять, почему так произошло, нам необходимо снова вспомнить теорему Пифагора из школьного курса, которая позволяет вычислять длины сторон прямоугольных треугольников. Согласно теореме, длина гипотенузы (самой длинной, диагональной стороны треугольника) равна $\sqrt{a^2 + b^2}$, где a и b — длины катетов. Для равностороннего прямоугольного треугольника с углами 45° формула принимает вид $\sqrt{2a^2} = a\sqrt{2}$. В случае с нашими диагональными полосами размер фона определяет длину гипотенузы, но ширина полосы в действительности равна длине катета. Схема, объясняющая эти расчеты, показана на рис. 2.32.

Это означает, что для того, чтобы получить полосы шириной `15px`, как в предыдущих примерах, в качестве размера фона необходимо указать значение $2 \times 15\sqrt{2} \approx 42{,}426406871$ пиксела:

```
background: linear-gradient(45deg,
            #fb3 25%, #58a 0, #58a 50%,
            #fb3 0, #fb3 75%, #58a 0);
background-size: 42.426406871px 42.426406871px;
```

Конечный результат показан на рис. 2.33. Но если только на вас не наставили дуло пистолета и не заставляют под страхом смерти создавать диагональные полосы шириной ровно 15 пикселов (кстати,

гибель все равно неизбежна, потому что $\sqrt{2}$ — нерациональное число, так что даже указанное нами значение приблизительное, хотя и с высокой степенью точности), я настоятельно рекомендую округлять это тяжеловесное число до чего-нибудь вроде `42.4px` или даже `42px`.

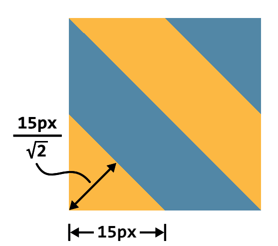


Рис. 2.32. Фон размером `30px` обеспечивает полосы шириной $\frac{15}{\sqrt{2}} \approx 10,606601718$ пиксела

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**diagonal-stripes**

Еще лучшие диагональные полосы



Рис. 2.33. Готовые полосы под наклоном 45°; обратите внимание, что ширина полос такая же, как в предыдущих примерах

Метод, продемонстрированный в предыдущем разделе, не обеспечивает особой гибкости. А что, если мы захотим создать полосы под углом 60°, а не 45°? Или 30°? Или 3,1415926535°? Если мы попытаемся изменить угол градиента, результат будет просто ужасным (на рис. 2.34 показана неудачная попытка нарисовать полосы под углом 60°).

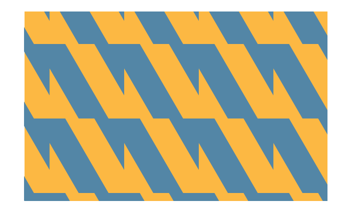


Рис. 2.34. Наша неудачная наивная попытка создания полос под углом 60°

К счастью, существует гораздо лучший способ рисования диагональных полос. Этот факт не слишком известен, но `linear-gradient()` и `radial-gradient()` также предлагают версии с повторением: `repeating-linear-gradient()` и `repeating-radial-gradient()`. Они работают совершенно так же, с одним только отличием: границы перехода цвета повторяются бесконечно, пока не заполнят все изображение. Так, например, следующий повторяющийся градиент (показанный на рис. 2.35):

```
background: repeating-linear-gradient(45deg,
            #fb3, #58a 30px);
```

эквивалентен этому простому линейному градиенту:

```
background: linear-gradient(45deg,
            #fb3, #58a 30px,
            #fb3 30px, #58a 60px,
            #fb3 60px, #58a 90px,
            #fb3 90px, #58a 120px,
            #fb3 120px, #58a 150px, ...);
```

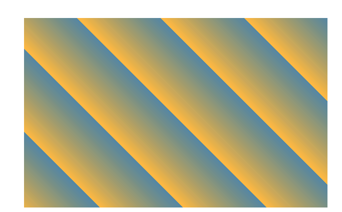


Рис. 2.35. Повторяющийся линейный градиент

Повторяющиеся линейные градиенты идеальны для — вы уже догадались — полос! Благодаря их повторяющейся природе мы можем создавать цельные фоны из сгенерированных градиентных изображений. Это означает, что нам больше не нужно беспокоиться о рисовании бесшовно стыкующихся плиток, которые затем нужно уложить для формирования фона элемента.

Для сравнения: фоновое изображение на рис. 2.33 можно было бы создать с помощью такого повторяющегося градиента:

```
background: repeating-linear-gradient(45deg,
            #fb3, #fb3 15px, #58a 0, #58a 30px);
```

Первое очевидное преимущество — уменьшение количества повторений: для того чтобы изменить любой из цветов, нам нужно внести только две правки вместо трех. Также обратите внимание, что теперь размеры определяются в терминах границ перехода цвета градиента, а не `background-size`. Размер фона используется первоначальный; для градиента это размер элемента. Это означает, что длины также становятся более понятными, так как измеряются они по *градиентной линии,* которая перпендикулярна нашим полосам. Больше никаких неуклюжих вычислений с $\sqrt{2}$!

Однако самое больше преимущество состоит в том, что теперь мы можем задать абсолютно любой угол, и градиент будет работать — не нужно больше размышлять над проработкой плиток с бесшовным соединением. Например, вот как определяются полосы под углом 60° (рис. 2.36):

```
background: repeating-linear-gradient(60deg,
            #fb3, #fb3 15px, #58a 0, #58a 30px);
```



Рис. 2.36. Настоящие полосы под углом 60°

Потребовалось всего лишь изменить угол! Обратите внимание, что с этим методом нам требуются четыре границы перехода цвета для двух цветных полос, независимо от угла наклона полос. Это означает, что для создания горизонтальных и вертикальных полос лучше все же использовать первый метод, а к этому прибегать для определения диагональных полос. Если же мы имеем дело с полосами под углом 45°, то эти два метода можно даже объединить, по сути, воспользовавшись повторяющимися линейными градиентами для упрощения кода, который создает нашу повторяющуюся плитку:

```
background: repeating-linear-gradient(45deg,
            #fb3 0, #fb3 25%, #58a 0, #58a 50%);
background-size: 42.426406871px 42.426406871px;
```

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**diagonal-stripes-60deg**

ПРОТЕСТИРУЙТЕ!

http://play.csssecrets.io/**test-color-stop-2positions**

Гибкие нежные полосы

Чаще всего нам требуется, чтобы полосы были не совершенно разных цветов, а представляли собой мягкие вариации яркости одного и того же цвета. Например, взгляните на такие полосы:

```
background: repeating-linear-gradient(30deg,
            #79b, #79b 15px, #58a 0, #58a 30px);
```

Как видно на рис. 2.37, мы создали чередующиеся полосы цвета `#58a` и одного из более светлых его вариантов. Однако взаимоотношение между этими цветами не очевидно, если просто прочитать код. Более того, если бы нам потребовалось изменить базовый цвет, то это повлекло бы за собой четыре (!) правки.

БУДУЩЕЕ.
ГРАНИЦЫ ПЕРЕХОДА ЦВЕТА С ДВУМЯ ПОЗИЦИЯМИ

Скоро у нас появится возможность указывать две позиции для одной и той же границы перехода цвета — это одно из базовых запланированных дополнений в **CSS Image Values Level 4** (http://w3.org/TR/css4-images). Это задумано как сокращение, позволяющее задать две последовательные границы перехода цвета с одним и тем же цветом, но разными позициями — очень востребованная функциональность при создании градиентных узоров. Например, код для диагональных полос с рис. 2.36 выглядел бы так:

```
background: repeating-linear-gradient(60deg,
            #fb3 0 15px, #58a 0 30px);
```

Это не только намного более емкий код, но и намного более соответствующий принципам DRY: цвета больше не дублируются, поэтому для изменения цвета достаточно одной правки. К сожалению, на момент написания этой главы данная функциональность не поддерживается ни одним браузером.

К счастью, существует лучший способ: вместо того чтобы задавать собственный цвет для каждой полосы, мы можем сделать наш самый темный цвет цветом фона, который будет просвечивать сквозь полосы полупрозрачного белого цвета:

```
background: #58a;
background-image: repeating-linear-gradient(30deg,
                    hsla(0,0%,100%,.1),
                    hsla(0,0%,100%,.1) 15px,
                    transparent 0, transparent 30px);
```



Рис. 2.37. Полосы с нежными вариациями яркости

Результат выглядит точно так же, как на рис. 2.37, но теперь для изменения цвета нам требуется внести правку только в одном месте. Также мы получаем дополнительное преимущество в том смысле, что наш базовый цвет будет служить резервным цветом в браузерах, не поддерживающих градиенты CSS. Помимо этого, как мы узнаем из следующего секрета, накладывая друг на друга градиентные узоры с полупрозрачными областями, можно создавать очень сложные рисунки.

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**subtle-stripes**

СВЯЗАННЫЕ СПЕЦИФИКАЦИИ

CSS Image Values: http://w3.org/TR/css-images

CSS Backgrounds & Borders: http://w3.org/TR/css-backgrounds

CSS Image Values Level 4: http://w3.org/TR/css4-images

6 Сложные фоновые узоры

ПРЕДВАРИТЕЛЬНЫЕ ТРЕБОВАНИЯ

Градиенты CSS, секрет «Фон в полоску»

Проблема

В предыдущем разделе мы узнали, как с помощью градиентов CSS создавать всевозможные полосы. Однако полосами фоновые узоры, да и любые другие геометрические рисунки, не ограничиваются. Часто у нас возникает необходимость создавать другие типы узоров: сетки, узор в горошек, шахматный узор и многие другие.

К счастью, градиенты CSS могут помочь и со многими из этих задач. С помощью **градиентов CSS можно создать почти любой геометрический узор**, хотя это и **не всегда практично**: если мы не будем соблюдать осторожность, то на руках у нас окажется безумный объем не поддающегося сопровождению кода. Создание узоров средствами CSS — это также одна из ситуаций, когда использование препроцессора CSS, например **Sass** (http://sass-lang.com), оправданно. Это позволяет сократить количество повторений, так как чем сложнее становятся узоры, тем менее соответствует принципам DRY описывающий их CSS-код.

В этом секрете мы сосредоточимся на создании самых простых и наиболее востребованных узоров.



Рис. 2.38. Моя галерея узоров, сделанных с помощью CSS3 (вы можете найти ее по адресу http://lea.verou.me/css3patterns), показывает, какие возможности обеспечивали градиенты CSS еще в 2011 году. В 2011–2012 годах на эту страницу ссылались почти в каждой статье и книге, в которых заходила речь о градиентах CSS, а также упоминали в выступлениях на множестве тематических конференций. Несколько производителей браузеров использовали ее для тонкой регулировки своих реализаций градиентов CSS. Однако далеко не все показанные узоры уместно использовать на реальных веб-сайтах. Некоторые из них приведены только для того, чтобы продемонстрировать возможности CSS, а код, необходимый для их создания, чрезвычайно объемен и полон повторений. Для подобных ситуаций намного лучше подходит формат SVG. Несколько примеров SVG-узоров вы найдете на http://philbit.com/svgpatterns; этот веб-сайт был создан в качестве ответа галерее узоров CSS

Сетки

Используя только один градиент, можно создать не так уж много узоров. Волшебство начинается, когда вы **сочетаете несколько градиентов**, просвечивающих сквозь друг друга благодаря наличию прозрачных областей. Вероятно, самый простой из подобных узоров — это наложение горизонтальных и вертикальных полос для создания различных типов сеток. Например, следующий фрагмент кода создает узор, напоминающий расцветку хлопчатобумажной скатерти (рис. 2.39):

```
background: white;
background-image: linear-gradient(90deg,
                    rgba(200,0,0,.5) 50%, transparent 0),
                  linear-gradient(
                    rgba(200,0,0,.5) 50%, transparent 0);
background-size: 30px 30px;
```

В некоторых случаях мы хотим иметь возможность **регулировать размер ячеек сетки, не меняя при этом ширину линий**, например, для создания линий, служащих в качестве направляющих. Это прекрасный повод использовать **абсолютные значения вместо процентных** в качестве границ перехода цвета:

```
background: #58a;
background-image:
    linear-gradient(white 1px, transparent 0),
    linear-gradient(90deg, white 1px,
                  transparent 0);
background-size: 30px 30px;
```

Результат (показанный на рис. 2.40) представляет собой сетку из белых линий шириной `1px`, где ширина ячейки равна `30px`. Так же как в секрете «Гибкие нежные полосы», базовый цвет служит резервным вариантом для случаев, когда браузер не поддерживает градиенты CSS.

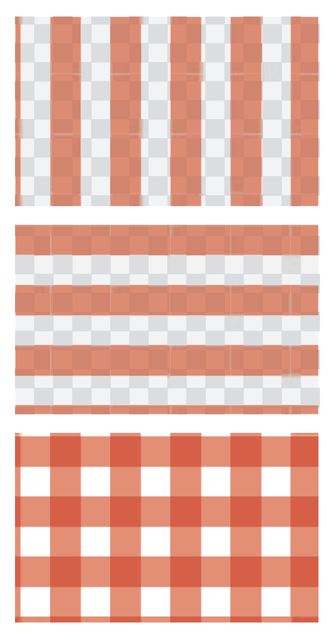


Рис. 2.39. Наш узор для скатерти, а также два градиента, из которых он состоит (здесь прозрачные области обозначены традиционной серой шахматной клеткой)

Эта сетка — отличный пример узора, который может быть создан с помощью достаточно хорошо поддающегося сопровождению CSS-кода (хотя и не полностью соответствующего принципам DRY):

- если нам понадобится поменять размер ячейки, толщину линий или любой из цветов, догадаться, какое значение требуется для этого отредактировать, будет довольно просто;

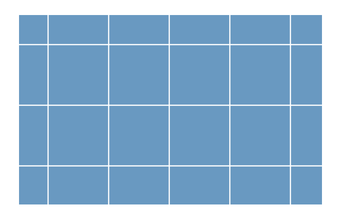


Рис. 2.40. Простейшая сетка для паттерна CSS, толщина линий на которой всегда остается равной `1px`, независимо от размера ячейки

- для внесения подобных изменений не приходится делать огромное количество правок; нужно исправить лишь одно или два значения;
- код к тому же относительно короткий, всего лишь четыре строки объемом 170 байт. SVG-код не мог бы быть короче.

Мы можем также наложить друг на друга две сетки с линиями разной толщины и разными цветами, для того чтобы создать более реалистичный вариант светокопировального листа (рис. 2.41):

```
background: #58a;
background-image:
    linear-gradient(white 2px, transparent 0),
    linear-gradient(90deg, white 2px, transparent 0),
    linear-gradient(hsla(0,0%,100%,.3) 1px,
        transparent 0),
    linear-gradient(90deg, hsla(0,0%,100%,.3) 1px,
        transparent 0);
background-size: 75px 75px, 75px 75px,
                 15px 15px, 15px 15px;
```

СОВЕТ

Для вычисления размера файла, содержащего код для вашего узора CSS, воспользуйтесь помощью http://bytesizematters.com — просто вставьте код в текстовое поле.

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**blueprint**

Узор в горошек

Пока что для создания узоров мы использовали только линейные градиенты. Однако радиальные градиенты тоже могут быть чрезвычайно полезными, так как они позволяют создавать окружности, эллипсы и фрагменты этих фигур. Самый простой узор, который можно создать с помощью радиального градиента, — это массив точек (рис. 2.42):

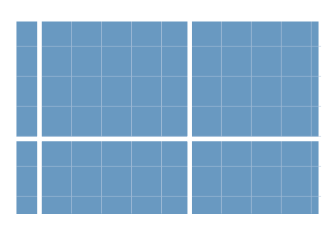


Рис. 2.41. Более сложная сетка для паттерна CSS, состоящая из двух сеток с разными параметрами

```
background: #655;
background-image: radial-gradient(tan 30%,
transparent 0);
background-size: 30px 30px;
```

Следует признать, что сам по себе этот рисунок не слишком пригоден для использования. Однако мы можем объединить два таких градиента и определить для них разные позиции фона, создав таким образом узор в горошек (рис. 2.43):

```
background: #655;
background-image: radial-gradient(tan 30%,
                  transparent 0),
                  radial-gradient(tan 30%,
                  transparent 0);
background-size: 30px 30px;
background-position: 0 0, 15px 15px;
```

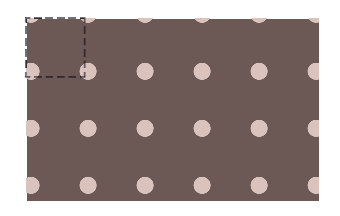


Рис. 2.42. Массив точек; повторяющаяся плитка обозначена пунктирной линией

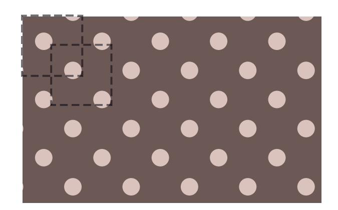


Рис. 2.43. Узор в горошек; обе повторяющиеся плитки обозначены пунктирными линиями

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**polka**

Обратите внимание: чтобы этот эффект сработал, вторая позиция фона должна составлять половину размера плитки. К сожалению, это означает, что для того, чтобы изменить размер плитки, нам нужно внести четыре правки. Это на грани того, чтобы назвать такой код непригодным к сопровождению, хотя общего мнения относительно того, перейдена ли черта, нет. Если вы используете препроцессор, то можете преобразовать это в примесь:

SCSS

```
@mixin polka($size, $dot, $base, $accent) {
    background: $base;
    background-image:
        radial-gradient($accent $dot, transparent 0),
        radial-gradient($accent $dot, transparent 0);
    background-size: $size $size;
    background-position: 0 0, $size/2 $size/2;
}
```

Затем для создания узора в горошек понадобится вызов, подобный этому:

SCSS

```
@include polka(30px, 30%, #655, tan);
```

Шахматные доски

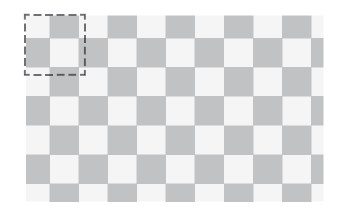


Рис. 2.44. Узор с серой шахматной доской для обозначения прозрачности; если бы мы создавали его путем повторения изображения, то для этого нам потребовалась бы плитка, обозначенная пунктирной линией

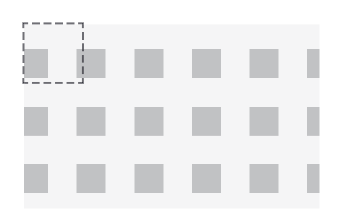


Рис. 2.45. Повторяющаяся плитка, на которой квадратик окружен пустым пространством; плитка обозначена пунктирной линией

Узоры типа «шахматная доска» используются во множестве ситуаций. Например, шахматная доска нежной расцветки может быть интересной альтернативой пресному фону сплошного цвета. Помимо этого, серая шахматная доска стала де факто стандартом обозначения прозрачности, что находит применение в самых разных пользовательских интерфейсах. Сделать шахматную доску с помощью CSS значительно сложнее, чем можно было бы ожидать.

Типичная плитка, повторение которой генерирует шахматную доску, включает по два квадратика каждого цвета, как показано на рис. 2.44. Кажется, что этот эффект должно быть несложно воссоздать с помощью CSS: нужно всего лишь определить два квадрата с разными позициями фона, не так ли? Не совсем. Да, технически возможно создать квадраты, используя градиенты CSS, но без пустого пространства вокруг них результат будет выглядеть как заливка сплошным цветом. Однако не существует способа создавать квадраты, окруженные пустым пространством, используя только один градиент CSS. Если вы считаете, что это не так, попытайтесь найти градиент, который при повторении будет создавать изображение, показанное на рис. 2.45.

Хитрость состоит в том, чтобы **делать квадратик из двух прямоугольных треугольников**. Мы уже умеем создавать прямоугольные треугольники (см. нашу неудачную попытку создать диагональные полосы на рис. 2.29). Вы можете освежить память, взглянув на следующий фрагмент кода (здесь используются другие цвета и прозрачность):

```
background: #eee;
background-image:
    linear-gradient(45deg, #bbb 50%,
                    transparent 0);
background-size: 30px 30px;
```

Возможно, вы задаетесь вопросом, чем это может вам помочь. Определенно, если попытаться сделать квадратики из двух треугольников, подобных показанным на рис. 2.29, результатом станет сплошной цвет. А что, если вполовину уменьшить катеты этих треугольников, чтобы они занимали 1/8 плитки, а не 1/2, как сейчас? Этого можно с легкостью добиться, **указав в качестве позиции границы перехода цвета 25% вместо 50%**. Результат будет выглядеть как на рис. 2.46.

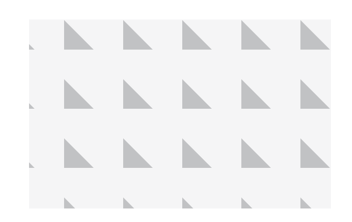


Рис. 2.46. Прямоугольные треугольники с большим количеством пустого пространства вокруг

Схожим образом можно создать треугольники, указывающие в противоположном направлении, зеркально отразив границы перехода цвета (рис. 2.47):

```
background: #eee;
background-image:
    linear-gradient(45deg, transparent 75%,
                    #bbb 0);
background-size: 30px 30px;
```

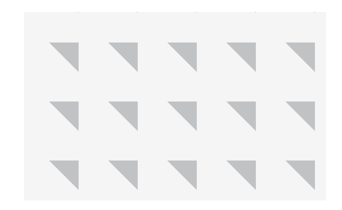


Рис. 2.47. Если зеркально отразить границы перехода цвета, то мы получим треугольники, указывающие в противоположном направлении

Угадаете, что произойдет, если мы объединим эти два решения? Код будет выглядеть так:

```
background: #eee;
background-image:
    linear-gradient(45deg, #bbb 25%,
                    transparent 0),
    linear-gradient(45deg, transparent 75%,
                    #bbb 0);
background-size: 30px 30px;
```

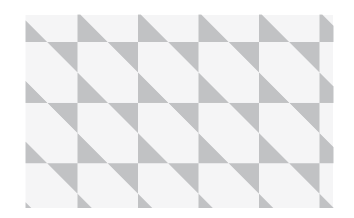


Рис. 2.48. Объединение двух треугольников

На первый взгляд кажется, что результат, показанный на рис. 2.48, не способен привести нас к желаемой цели. Однако нужно всего лишь **сдвинуть второй градиент на половину размера плитки**, для того чтобы объединить эти треугольники и получить квадрат:

```
background: #eee;
background-image:
    linear-gradient(45deg, #bbb 25%, transparent 0),
    linear-gradient(45deg, transparent 75%, #bbb 0);
background-position: 0 0, 15px 15px;
background-size: 30px 30px;
```

БУДУЩЕЕ.
КОНИЧЕСКИЕ ГРАДИЕНТЫ



В будущем для создания шахматных досок нам не придется полагаться исключительно на помощь треугольников, педантично накладываемых друг на друга. CSS Image Values Level 4 (http://w3.org/TR/css4-images) определяет новый набор функций градиента, позволяющих создавать конические градиенты (также известные как «угловые градиенты»). Эти градиенты часто выглядят как конусы, если смотреть на них сверху, отсюда и название. Они генерируются с помощью линии, которая крутится вокруг фиксированной точки, постепенно меняя цвет. Например, цветовое колесо, показанное здесь, можно будет создать с помощью следующего градиента:

```
background: conic-gradient(red, yellow, lime, aqua, blue,
            fuchsia, red);
```

Конические градиенты удобны для определения множества различных эффектов, не только для создания цветового колеса: звездные взрывы, эффект выглаженного щеткой металла и многие другие типы фонов, включая (как вы уже догадались) шахматные доски. Благодаря им повторяющуюся плитку с рис. 2.44 можно было бы создать с помощью всего лишь одного градиента:

```
background: repeating-conic-gradient(#bbb 0, #bbb 25%, #eee 0,
            #eee 50%);
background-size: 30px 30px;
```

К сожалению, на момент написания этой главы конические градиенты не поддерживаются ни в одном браузере, но вы найдете полифилл на веб-странице http://leaverou.github.io/conic-gradient.

ПРОТЕСТИРУЙТЕ!

http://play.csssecrets.io/**test-conic-gradient**

Догадаетесь, каким будет результат? Это как раз то, чего мы пытались добиться ранее, — см. рис. 2.49. Обратите внимание, что, по сути, это **половинчатая шахматная доска**. Все, что нам нужно для превращения ее в полноценную шахматную доску, — это повторить два градиента, создав еще один набор квадратиков, и еще раз сместить их позиции, как если бы мы дважды применяли технику создания узора в горошек:

```
background: #eee;
background-image:
    linear-gradient(45deg, #bbb 25%, transparent 0),
    linear-gradient(45deg, transparent 75%, #bbb 0),
    linear-gradient(45deg, #bbb 25%, transparent 0),
    linear-gradient(45deg, transparent 75%, #bbb 0);
background-position: 0 0, 15px 15px,
                     15px 15px, 30px 30px;
background-size: 30px 30px;
```

Результатом станет шахматная доска, идентичная показанной на рис. 2.44. Мы можем немного усовершенствовать код, объединив треугольники, указывающие в противоположные стороны (то есть первый со вторым, а третий с четвертым) и превратив более темный оттенок серого в полупрозрачный черный, для того чтобы базовый цвет можно было всегда с легкостью поменять без необходимости соответствующим образом корректировать цвет верхнего слоя:

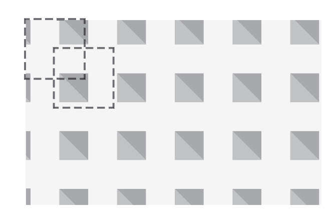


Рис. 2.49. Теперь наши объединенные треугольники формируют квадратики, окруженные пустым пространством; две плитки обозначены пунктирными линиями, а для второго градиента используется чуть более темный оттенок

```
background: #eee;
background-image:
    linear-gradient(45deg,
        rgba(0,0,0,.25) 25%, transparent 0,
        transparent 75%, rgba(0,0,0,.25) 0),
    linear-gradient(45deg,
        rgba(0,0,0,.25) 25%, transparent 0,
        transparent 75%, rgba(0,0,0,.25) 0);
background-position: 0 0, 15px 15px;
background-size: 30px 30px;
```

Теперь у нас два градиента вместо четырех, но код, как и раньше, может служить иллюстрацией принципа WET. Для того чтобы изменить акцентный цвет или размер ячейки, необходимо внести четыре правки. В данном случае уместно создать примесь в препроцессоре, чтобы исключить дублирование. Например, в Sass это выглядело бы так:

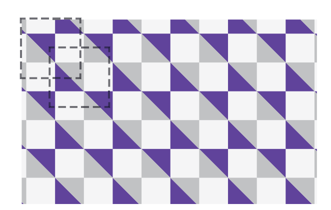


Рис. 2.50. Это сложный узор, и догадаться, как он работает, особенно после уменьшения количества градиентов до двух, не так-то легко. Чаще всего разобраться в хитросплетениях узора помогает присвоение случайного цвета одному из градиентов или одной из границ перехода цвета. Например, здесь первый градиент показан цветом `rebeccapurple` вместо полупрозрачного черного, а две плитки обозначены пунктирными линиями

SCSS

```
@mixin checkerboard($size, $base,
                    $accent: rgba(0,0,0,.25)) {
background: $base;
background-image:
    linear-gradient(45deg,
        $accent 25%, transparent 0,
        transparent 75%, $accent 0),
    linear-gradient(45deg,
        $accent 25%, transparent 0,
        transparent 75%, $accent 0);
    background-position: 0 0, $size $size,
    background-size: 2*$size 2*$size;
}

/* Пример использования */
@include checkerboard(15px, #58a, tan);
```

В любом случае, кода получилось так много, что, возможно, было бы лучше пойти по пути SVG. Плитка для рис. 2.44 в формате SVG была бы очень простой и короткой:

SVG

```
<svg xmlns="http://www.w3.org/2000/svg"
    width="100" height="100" fill-opacity=".25"
>
    <rect x="50" width="50" height="50" />
    <rect y="50" width="50" height="50" />
</svg>
```

WET расшифровывается как *We Enjoy Typing* («Нам нравится печатать»), и это противоположность принципу DRY (то есть WET-код — это повторяющийся, не поддающийся нормальному сопровождению код).

Кто-то возразит: «Но градиенты CSS экономят нам HTTP-запросы!» Однако в современных браузерах мы можем встроить SVG-файл в таблицу стилей как URI с данными, и нам даже почти не придется преобразовывать его с помощью base64 или URLencode:

```
background: #eee url('data:image/svg+xml,\
    <svg xmlns="http://www.w3.org/2000/svg" \
    width="100" height="100"
    fill-opacity=".25">\
    <rect x="50" width="50" height="50" /> \
    <rect y="50" width="50" height="50" /> \
    </svg>');
background-size: 30px 30px;
```

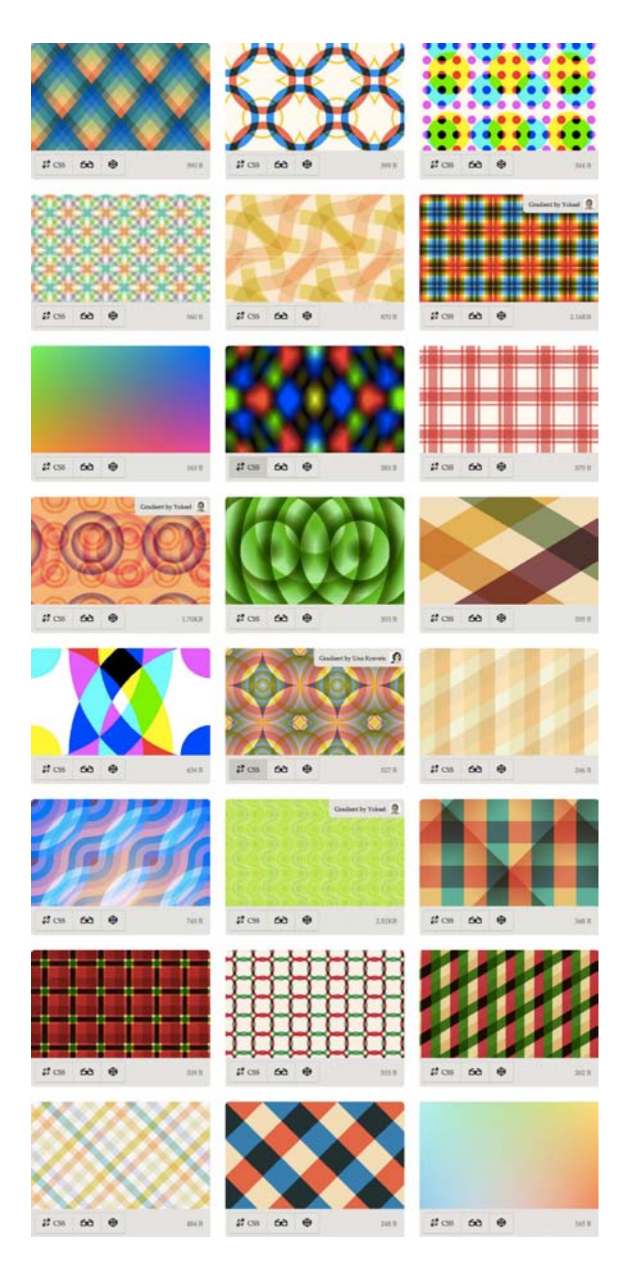


Рис. 2.51. Сочетание этих техник с **режимами смешивания** (http://w3.org/TR/compositing-1), когда значения background-blend-mode, отличные от normal, используются для некоторых (или даже всех) слоев, составляющих фоновое изображение, может дать очень интересные результаты, что подтверждает эта галерея узоров Беннетта Фили (http://bennettfeely.com/gradients). Для большинства из этих узоров используется только режим смешивания multiply, но и другие значения, такие как overlay, screen и difference, также могут оказаться очень полезными

СОВЕТ

Обратите внимание, что одну строку CSS-кода для удобства чтения можно разнести на несколько строк файла, добавив перед символом перевода строки обратный слеш (\).

Версия в формате SVG не только на 40 символов короче, в ней также заметно меньше повторений. Например, для изменения цвета достаточно одной правки, а для изменения размера — двух.

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**checkerboard-svg**

СВЯЗАННЫЕ СПЕЦИФИКАЦИИ

CSS Image Values: http://w3.org/TR/css-images

CSS Backgrounds & Borders: http://w3.org/TR/css-backgrounds

Scalable Vector Graphics: http://w3.org/TR/SVG

CSS Image Values Level 4: http://w3.org/TR/css4-images

7 (Псевдо)случайные фоны

ПРЕДВАРИТЕЛЬНЫЕ ТРЕБОВАНИЯ

Градиенты CSS, секрет «Фон в полоску», секрет «Сложные фоновые узоры»

Проблема

Повторяющиеся геометрические узоры смотрятся мило, но бывают немного скучными. **Вряд ли где-то в природе можно встретить узор, состоящий из идентичных повторяющихся плиток**. Даже когда рисунок повторяется, он все равно содержит массу вариаций и случайностей. Взгляните на цветочное поле: хотя оно достаточно однородно, чтобы казаться нам красивым, все равно там присутствует достаточно бессистемности, чтобы вызывать интерес. Невозможно найти два абсолютно одинаковых цветка. Вот почему, когда мы пытаемся создавать фоновые узоры, выглядящие как можно более естественными, мы одновременно стараемся, чтобы «швов» между повторяющимися плитками было как можно меньше и они были как можно менее заметными, а это прямо противоречит нашему желанию сохранять небольшой размер файла.



Рис. 2.52. Природа не повторяет себя в «бесшовных» плитках

Когда характерная черта — например, завиток в текстуре древесины — повторяется через равные интервалы, это сразу разрушает иллюзию природной случайности.

Алекс Уолкер, Принцип цикады и почему это важно для веб-дизайнеров (Alex Walker, The Cicada Principle and Why It Matters to Web Designers) (http://sitepoint.com/the-cicada-principle-and-why-it-matters-to-web-designers)

Воспроизвести случайность — задача непростая, так как в CSS не предусмотрено никаких встроенных возможностей генерации случайных значений. Рассмотрим пример с полосами. Предположим, что мы хотим создать вертикальные полосы разных цветов и ширины (для простоты ограничимся четырьмя цветами) без каких-либо видимых «швов» между повторяющимися плитками. Первой мыслью может быть создание одного градиента со всеми четырьмя полосами, например, так:

```
background: linear-gradient(90deg,
            #fb3 15%, #655 0, #655 40%,
            #ab4 0, #ab4 65%, hsl(20, 40%, 90%) 0);
background-size: 80px 100%;
```



Рис. 2.53. Наша первая попытка создать псевдослучайные полосы, где все цвета генерируются одним и тем же линейным градиентом

Как видно на рис. 2.53, повторения очевидны, так как шаблон воспроизводится каждые `80px` (это значение `background-size`). Можно ли добиться чего-то лучшего?

Решение

Первая идея — создать иллюзию случайности, разбив плоскую полосатую плитку на два слоя: один базовый слой и три слоя полосок, повторяющихся с разными интервалами. Это легко реализовать, жестко закодировав ширину полос в границах перехода цвета и используя `background-size` для управления расстоянием между полосами. Код может выглядеть примерно так:

```
background: hsl(20, 40%, 90%);
background-image:
    linear-gradient(90deg, #fb3 10px, transparent 0),
    linear-gradient(90deg, #ab4 20px, transparent 0),
    linear-gradient(90deg, #655 20px, transparent 0);
background-size: 80px 100%, 60px 100%, 40px 100%;
```

Так как повторение самой верхней плитки будет наиболее заметно (ведь ее ничто не заслоняет), **наверх следует поместить плитку с самым большим интервалом повторения** (в нашем случае это оранжевые полоски).

Обратите внимание, что здесь слово «плитка» используется в абстрактном смысле: мы говорим не о повторяющемся изображении каждого отдельного градиента, а о **воспринимаемой взглядом повторяющейся плитке, представляющей собой композицию градиентов** (то есть если бы мы не использовали решение с несколькими фонами, то какого размера повторяющееся фоновое изображение потребовалось бы нам для того, чтобы добиться того же результата?).

Как демонстрирует рис. 2.54, теперь результат намного более похож на случайный набор полосок, но если приглядеться, все же можно заметить, что одна и та же плитка повторяется каждые `240px`. Конец первой повторяющейся плитки для такой композиции приходится на точку, до которой **все наши отдельные фоновые изображения были воспроизведены целое число раз**. Как вы помните из школьного курса математики, если у нас есть несколько чисел, то минимальное число, которое нацело делится на каждое из них, — это их наименьшее общее кратное (часто это название сокращают до аббревиатуры НОК). Следовательно, здесь **размер плитки — это наименьшее общее кратное размеров фона**, то есть НОК для 40, 60 и 80, и это значение равно 240.

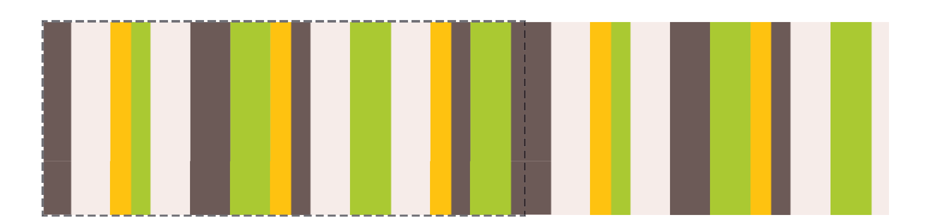


Рис. 2.54. Наша вторая попытка, включающая наложение друг на друга разных градиентов с разным размером фона; (воспринимаемая взглядом) повторяющаяся плитка обозначена пунктирными линиями

Отсюда логически вытекает, что для того, чтобы сделать узор еще более визуально хаотичным, необходимо **максимизировать размер повторяющейся плитки**. Благодаря математике нам не приходится долго и тяжело размышлять, как же это сделать, так как ответ уже известен. **Чтобы НОК был максимальным**,

числа должны быть взаимно простыми.[1] В этом случае их НОК будет равно их произведению. Например, 3, 4 и 5 взаимно простые, поэтому их НОК равно 3 × 4 × 5 = 60. Самый простой способ подобрать подходящие значения — воспользоваться **простыми числами, так как они всегда взаимно просты с любыми другими числами**. Списки простых чисел вплоть до очень больших значений легко можно найти на различных веб-сайтах в Сети.



Рис. 2.55. Финальный вариант полосок, где для создания лучшего впечатления хаотичности используются простые числа

Чтобы создать еще больший эффект случайности, для указания ширины полос можно также использовать простые числа. Вот как будет выглядеть новая версия кода:

```
background: hsl(20, 40%, 90%);
background-image:
    linear-gradient(90deg, #fb3 11px, transparent 0),
    linear-gradient(90deg, #ab4 23px, transparent 0),
    linear-gradient(90deg, #655 41px, transparent 0);
background-size: 41px 100%, 61px 100%, 83px 100%;
```

Да, код не самый красивый, но попробуйте найти швы на рис. 2.55! Размер повторяющейся плитки теперь равен 41 × 61 × 83 = 207 583 пикселов — больше, чем любое разрешение экрана, какое только можно вообразить!

Эту технику Алекс Уолкер, который впервые догадался использовать простые числа для создания впечатления случайно сгенерированного фона, назвал **«Принципом цикады»**. Обратите внимание, что она может пригодиться не только для фонов, но и для любых других элементов, с которыми хорошо работают повторения. Варианты использования включают:

[1] Простые числа — это целые числа, **которые не делятся без остатка ни на какие другие числа, кроме 1 и самих себя**. Например, первые 10 простых чисел — это 2, 3, 5, 7, 11, 13, 17, 19, 23, 29. С другой стороны, «взаимно простые» относится к **взаимосвязи между числами**, то есть это не характеристика отдельного числа. У взаимно простых чисел нет общих делителей, но в целом делители у них присутствовать могут (например, 10 и 27 — взаимно простые, но ни одно из них простым не является). **Разумеется, простое число будет взаимно простым с любым другим числом.**

- небольшие псевдослучайные повороты изображений в фотогалерее с несколькими `:nth-child(an)`, где `a` — простое число;
- создание анимации, в которой фрагменты не повторяются всегда одним и тем же образом. Используйте несколько анимированных изображений, длительность которых равна простым числам (пример вы найдете на http://play.csssecrets.io/cicanimation).

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**cicada-stripes**


Благодарности

Спасибо Алексу Уолкеру за идею, послужившую вдохновением для этой техники. Подробное описание вы найдете на странице http://sitepoint.com/the-cicada-principle-and-why-it-matters-to-web-designers. Эрик Мейер (Eric Meyer) (http://meyerweb.com) позднее придумал нечто под названием «Цикадиенты» (*Cicadients* — http://meyerweb.com/eric/thoughts/2012/06/22/cicadients), подразумевающее применение этой техники к фоновым изображениям, сгенерированным с помощью градиентов CSS. Дадли Стори (Dudley Storey) также написал **очень информативную статью об этой концепции:** http://demosthenes.info/blog/840/Brood-X-Visualizing-The-Cicada-Principle-In-CSS.

8 Сплошные рамки для изображений

ПРЕДВАРИТЕЛЬНЫЕ ТРЕБОВАНИЯ

Градиенты CSS, базовые знания о `border-image`, секрет «Фон в полоску», базовые знания об анимации CSS



Рис. 2.56. Изображение камня, с которым мы будем работать в этом секрете

Проблема

Иногда у нас возникает необходимость использовать узор или изображение **не в качестве фона, а в качестве рамки**. Например, на рис. 2.57 вы видите элемент с декоративной рамкой, по сути, представляющей собой изображение, обрезанное так, чтобы от него осталась только рамка. Кроме того, мы хотим, чтобы наше изображение могло масштабироваться, закрывая всю площадь рамки, независимо от размеров элемента. Как создать нечто подобное с помощью CSS?

Возможно, у вас в голове сейчас звучит громкий крик: «`border-image`, `border-image`, мы можем использовать `border-image`, это больше не проблема!!!11» **Не так быстро, юный падаван**. Вспомните, как на самом деле работает `border-image`: по сути, это **масштабирование девяти фрагментов**. Вы разрезаете изображение на девять блоков и применяете их к углам и сторонам соответственно. На рис. 2.58 вы найдете визуальное напоминание того, как это работает.

Как с помощью `border-image` нарезать изображение, чтобы воспроизвести пример с рис. 2.57? Даже если мы потратим кучу времени и правильно разделим его для элемента конкретного размера с конкретной шириной рамки, оно не будет масштабироваться при изменении размера элемента. Проблема в том,

что мы не собираемся всегда использовать в углах лишь определенные части изображения; то, какой фрагмент изображения будет отображаться в углу, зависит от размера элемента и ширины рамки. Попробуйте поэкспериментировать, и, скорее всего, вы поймете, что с `border-image` это невозможно. Но что же тогда делать?

Самый простой способ — использовать два элемента HTML: один, для которого фоном будет служить наша картинка с камнем, и второй — с белым фоном для области содержимого элемента:

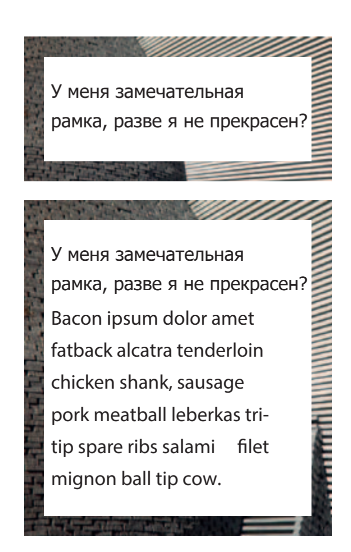


Рис. 2.57. Наше изображение используется в качестве рамки варьирующейся высоты

HTML

```
<div class="something-meaningful"><div>
    I have a nice stone art border,
    don't I look pretty?
</div></div>

.something-meaningful {
    background: url(stone-art.jpg);
    background-size: cover;
    padding: 1em;
}

.something-meaningful > div {
    background: white;
    padding: 1em;
}
```

Это решение хорошо работает и создает «рамку», показанную на рис. 2.57, но требует наличия дополнительного элемента HTML. Таким образом, оно не оптимально: оно не только смешивает представление и структуру, но в определенных случаях изменить HTML-код вообще невозможно. Существует ли способ реализовать то же самое с помощью одного элемента?

Решение

Благодаря градиентам CSS и расширениям для фона, представленным в **Backgrounds & Borders Level 3** (http://w3.org/TR/css3-background), мы можем достичь того же самого эффекта с одним элементом. Главная идея заключается в использовании второго фона чисто-белого цвета, который будет закрывать изображение с камнем. Однако для того, чтобы изображение камня проглядывало сквозь область рамки, к этим двум изображениям необходимо применить разные

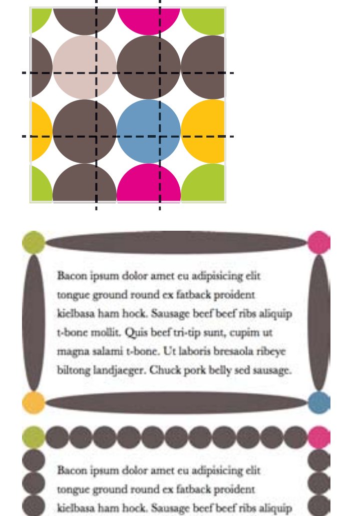


Рис. 2.58. Маленький урок о принципах работы `border-image`

Наверху: наше нарезанное изображение; пунктирными линиями обозначены линии разреза

Посередине: `border-image: 33.34% url(…) stretch;`

Внизу: `border-image: 33.34% url(…) round;`

Вы можете поэкспериментировать с кодом на http://play.csssecrets.io/border-image

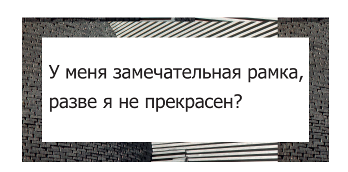


Рис. 2.59. В нашей первой попытке мы очень близки к успеху

значения `background-clip`. И последнее, о чем нужно помнить: мы можем использовать только фоновый цвет последнего слоя, поэтому нам потребуется имитировать белый фон с помощью градиента CSS от белого до белого цвета.

Вот как могла бы выглядеть первая попытка воплотить эту идею:

```
padding: 1em;
border: 1em solid transparent;
background: linear-gradient(white, white),
            url(stone-art.jpg);
background-size: cover;
background-clip: padding-box, border-box;
```

Как видно на рис. 2.59, результат очень близок к тому, чего мы хотели достичь, но все же наблюдается какое-то странное повторение. Причина в том, что значение по умолчанию для `background-origin` равно `padding-box`, и поэтому размер изображения вычисляется в зависимости от размера области забивки, а также помещается в точку 0, 0 относительно области забивки. Все остальное — это всего лишь повторения первой фоновой плитки. Чтобы исправить ситуацию, необходимо также установить значение `border-box` для свойства `background-origin`:

```
padding: 1em;
border: 1em solid transparent;
background: linear-gradient(white, white),
            url(stone-art.jpg);
background-size: cover;
background-clip: padding-box, border-box;
background-origin: border-box;
```

Эти новые свойства также доступны в сокращении `background`, которое способно помочь нам здорово уменьшить объем кода:

```
padding: 1em;
border: 1em solid transparent;
background:
    linear-gradient(white, white) padding-box,
    url(stone-art.jpg) border-box 0 / cover;
```

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**continuous-image-borders**

Разумеется, ту же технику можно применять и с **узорами, основанными на градиентах**. Например, взгляните на следующий код, в котором мы генерируем **рамку в стиле старомодного конверта**:

```
padding: 1em;
border: 1em solid transparent;
background: linear-gradient(white, white)
            padding-box,
            repeating-linear-gradient(-45deg,
                red 0, red 12.5%,
                transparent 0, transparent 25%,
                #58a 0, #58a 37.5%,
                transparent 0, transparent 50%)
            0 / 5em 5em;
```



Рис. 2.60. Настоящий старомодный конверт

Результат показан на рис. 2.61. Ширину полос можно с легкостью поменять с помощью свойства `background-size`, а ширина рамки регулируется объявлением `border`. В отличие от нашего примера с изображением камня, этот эффект также **можно воплотить с помощью** `border-image`:

```
padding: 1em;
border: 16px solid transparent;
border-image: 16 repeating-linear-
              gradient(-45deg,
                red 0, red 1em,
                transparent 0, transparent 2em,
                #58a 0, #58a 3em,
                transparent 0, transparent 4em);
```

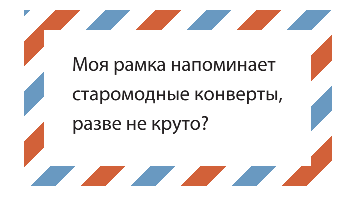


Рис. 2.61. Наша рамка в стиле старомодного конверта

СОВЕТ

Для того чтобы увидеть это в действии, зайдите на http://play.csssecrets.io/vintage-envelope-border-image и поэкспериментируйте с изменением значений.

Однако подход с `border-image` влечет за собой несколько проблем:

- значение `border-image-slice` необходимо обновлять каждый раз, когда мы меняем `border-width`, для того чтобы они соответствовали друг другу;
- так как с `border-image-slice` невозможно использовать значения, выраженные в `em`, мы **ограничены исключительно пикселами** при определении толщины рамки;
- толщину полос необходимо кодировать в позициях границ перехода цвета, поэтому для ее изменения потребуется внести четыре правки.

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**vintage-envelope**

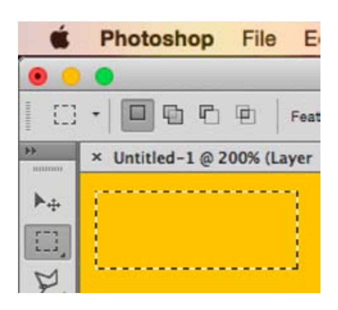


Рис. 2.62. Марширующие муравьи также используются в Adobe Photoshop для обозначения выделенной области

Еще одно интересное применение данной техники — создание **рамок из марширующих муравьев**! Рамки из марширующих муравьев — это пунктирные рамки, переливающиеся так, что создается впечатление, будто по периметру бежит цепочка муравьев. Они чрезвычайно часто встречаются в графических интерфейсах пользователя; в графических редакторах они практически повсеместно используются для обозначения выделенных областей (рис. 2.62).

Для того чтобы создать марширующих муравьев, мы воспользуемся одной из вариаций эффекта «старомодного конверта». Мы сделаем полосы черными и белыми, уменьшим ширину рамки до `1px` (вы заметили, что полосы уже превратились в пунктирную рамку?) и заменим `background-size` чем-нибудь более подходящим. Затем мы анимируем `background-position` до `100%`, чтобы создать переливающийся эффект:

```
@keyframes ants { to { background-position: 100% } }

.marching-ants {
    padding: 1em;
    border: 1px solid transparent;
    background:
        linear-gradient(white, white) padding-box,
        repeating-linear-gradient(-45deg,
            black 0, black 25%, white 0, white 50%
        ) 0 / .6em .6em;
    animation: ants 12s linear infinite;
}
```

Результат вы можете видеть на рис. 2.63. Очевидно, этот трюк полезен не только для имитации марширующих муравьев, но и для **создания всевозможных необычных пунктирных рамок, например многоцветных и с промежутками нестандартной величины между черточками**.

Единственный способ добиться схожего эффекта с помощью `border-image` — использовать анимированное изображение в формате GIF в качестве `border-image-source`, как демонстрирует пример на странице http://chrisdanford.com/blog/2014/04/28/marching-ants-animated-selection-rectangle-in-css. Когда в браузерах появится поддержка градиентной интерполяции, это можно будет делать и посредством градиентов, хотя и грубым, хаотичным путем с большим количеством WET-кода.

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**marching-ants**

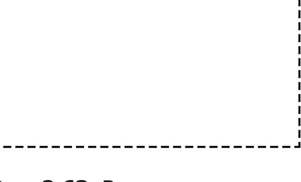


Рис. 2.63. В книге, разумеется, невозможно показать марширующих муравьев (стоп-кадр выглядит как обычная пунктирная рамка); зайдите на страницу с анимированным примером — это весело!

Однако свойство `border-image` также обладает большой мощью и может быть весьма полезным в сочетании с градиентами. Например, предположим, что нам требуется только фрагмент верхнего края рамки, скажем, для оформления сноски. Все, что нам для этого нужно, — это `border-image` и вертикальный градиент, в котором жестко закодирована точка обрезки рамки. Шириной рамки управляет... `border-width`. Соответствующий код мог бы выглядеть так:

```
border-top: .2em solid transparent;
border-image: 100% 0 0 linear-gradient(90deg,
                        currentColor 4em,
                        transparent 0);
padding-top: 1em;
```

[1] This is a footnote.

Рис. 2.64. Обрезка верхнего края рамки для имитации традиционной сноски

Результат идентичен показанному на рис. 2.64. Помимо этого, так как мы указали все значения в единицах `em`, этот эффект будет подстраиваться под изменения значения `font-size`. А благодаря использованию `currentColor` он также будет адаптироваться к изменениям `color` (в предположении, что мы хотим, чтобы рамка была того же цвета, что и текст).

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**footnote**

СВЯЗАННЫЕ СПЕЦИФИКАЦИИ

CSS Backgrounds & Borders: http://w3.org/TR/css-backgrounds

CSS Image Values: http://w3.org/TR/css-images

3

Фигуры

9 Гибкие эллипсы

ПРЕДВАРИТЕЛЬНЫЕ ТРЕБОВАНИЯ

Базовые навыки использования свойства `border-radius`

Проблема

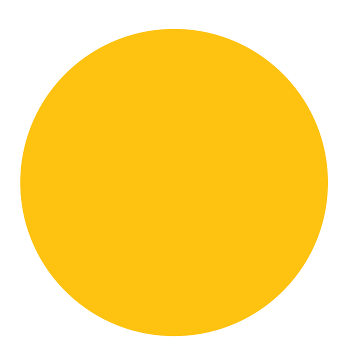


Рис. 3.1. Круг, сгенерированный путем указания фиксированных габаритных размеров квадрата и значения `border-radius`, равного половине длины его стороны

Вероятно, вы замечали, что любой квадратный элемент, для которого определено достаточно большое значение `border-radius`, можно превратить в круг с помощью примерно такого CSS-кода:

```
background: #fb3;
width: 200px;
height: 200px;
border-radius: 100px; /* >= половины длины
               стороны */
```

Возможно, вы также замечали, что в подобной ситуации можно было бы указать **любое** значение радиуса больше `100px` и все равно получить в результате круг. Причина объясняется в спецификации:

> *Если сумма любых двух радиусов соседних рамок превышает размер поля рамки, пользовательские агенты должны пропорционально уменьшать используемые значениях всех радиусов рамки, чтобы наложения не происходило.*
>
> **CSS Backgrounds & Borders Level 3**
> (http://w3.org/TR/css3-background/#corneroverlap)

Однако часто бывает так, что указать точные значения ширины и высоты элемента невозможно, так как мы хотим, чтобы он **подстраивался под свое содержимое**, что может быть неизвестно наперед. Даже если мы проектируем статичный веб-сайт и его точное содержимое определено заранее, не исключено, что в какой-то момент мы захотим модифицировать его или он будет отображаться с использованием резервного шрифта с другими метриками. В этом случае мы хотим, чтобы на выходе получался **эллипс в ситуации, когда ширина и высота не равны между собой, и круг, когда они совпадают**. Однако наш предыдущий код не способен обработать такой сценарий. Результирующая фигура для случая, когда ширина больше высоты, показана на рис. 3.2. Можно ли в принципе создавать эллипсы с помощью `border-radius`, не говоря уже о том, чтобы делать их гибкими?

Рис. 3.2. Наш предыдущий пример создания круга в ситуации, когда высота меньше ширины; круг, определяемый значением `border-radius`, обозначен пунктирной линией

Решение

Этот факт не слишком известен, но свойство `border-radius` способно принимать **разные значения для горизонтального и вертикального радиуса**, разделенные слешем (/). Это позволяет создавать **эллиптическое скругление** в углах элемента (рис. 3.3). Таким образом, если у нас есть элемент размером, скажем, `200px × 150px`, то мы могли бы превратить его в эллипс, указав радиусы, равные половине ширины и высоты соответственно:

Рис. 3.3. Поле с разными значениями горизонтального и вертикального радиуса в свойстве `border-radius`; скругление угла теперь повторяет контур эллипса, горизонтальный и вертикальный радиусы которого равны значениям из `border-radius`. Сам эллипс обозначен на рисунке пунктирной линией

```
border-radius: 100px / 75px;
```

Результат вы можете видеть на рис. 3.4.

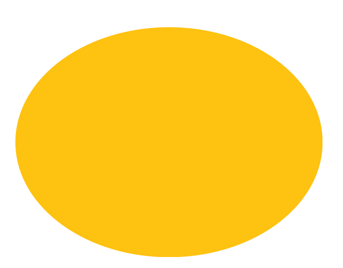


Рис. 3.4. Неравные параметры кривых, определяемые с помощью `border-radius`, позволяют создавать эллипсы

Однако у этого подхода есть один **крупный недостаток**: если габаритные размеры элемента меняются, то и значения `border-radius` также требуют корректировки. На рис. 3.5 вы можете видеть, как то же определение свойства `border-radius` работает с элементом размером `200px × 300px`. Габариты элемента, меняющиеся в зависимости от содержимого, создают проблему.

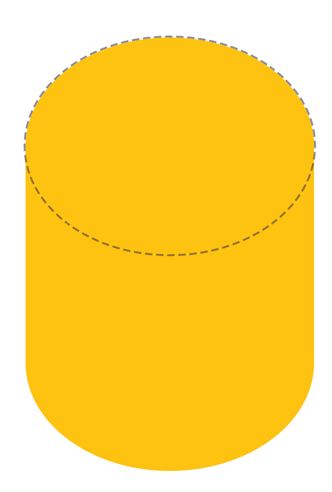


Рис. 3.5. Изменение габаритных размеров элемента ломает наш эллипс; но есть и хорошие новости — эта фигура будет незаменима, если нам понадобится нарисовать цилиндр!

Или нет? У `border-radius` есть еще одна малоизвестная особенность: это свойство принимает **не только абсолютные, но и процентные значения. Процентное значение разрешается в соответствующее габаритное значение**: горизонтальный радиус определяет ширину, а вертикальный радиус определяет высоту. Это означает, что **одни и те же процентные значения могут превращаться в разные значения горизонтального и вертикального радиуса**. Следовательно, для того чтобы получить гибкий эллипс, следует заменить оба радиуса значением `50%`:

```
border-radius: 50% / 50%;
```

И так как составляющие до и после слеша теперь равны (несмотря на то, что в итоге они разрешаются в разные значения), мы можем дополнительно упростить код:

```
border-radius: 50%;
```

Результат — гибкий эллипс, определяемый всего лишь одной строкой CSS-кода, независимо от его фактической ширины и высоты.

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**ellipse**

ЗАНИМАТЕЛЬНАЯ СТРАНИЧКА.
ПОЧЕМУ BORDER-RADIUS?

Многие задаются вопросом, почему свойство `border-radius` получило именно такое название, ведь для того, чтобы оно работало, никакие рамки (*border*) не требуются. Казалось бы, `corner-radius` было бы намного более подходящим названием. Причина существования такого (заведомо сбивающего с толку) названия заключается в том, что `border-radius` скругляет углы *поля рамки* элемента. Если у элемента нет рамок, то это ни на что не влияет, но если рамки имеются, то скругляется внешний угол именно рамки. Внутренний угол подвергается меньшему скруглению (`max(0, border-radius - border-width)`, если быть точными).

Полуэллипсы

Теперь, когда мы знаем, как с помощью CSS создавать гибкие эллипсы, естественным образом возникает вопрос: можно ли посредством CSS-кода рисовать и другие распространенные фигуры, такие как **дуги эллипса**? Давайте на мгновение задумаемся, что же такое половина эллипса (например, как та, что показана на рис. 3.6).

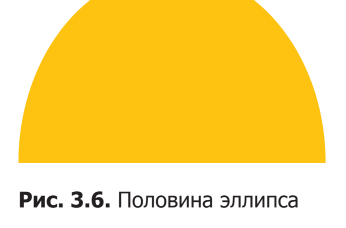


Рис. 3.6. Половина эллипса

Она симметрична относительно вертикальной оси, но не относительно горизонтальной оси. И даже если нам пока неизвестны точные значения `border-radius` (при условии, что их вообще можно узнать), уже очевидно, что нам потребуются разные значения радиуса для каждого из углов. Однако пока что мы умеем задавать лишь одно значение для всех четырех углов.

Половина эллипса может стать полукругом, если ее ширина будет в два раза превышать высоту (или если высота будет в два раза больше ширины — для эллипса, разрезанного по вертикальной оси).

К счастью, синтаксис `border-radius` намного более гибкий. Возможно, вы удивитесь, но `border-radius` — это в действительности сокращение. На самом деле мы можем указать разные значения для всех четырех углов, к тому же сделать это можно двумя способами. Первый способ — использовать полную запись свойств, составляющих это сокращение:

- `border-top-left-radius`
- `border-top-right-radius`
- `border-bottom-right-radius`
- `border-bottom-left-radius`

Второй способ позволяет создавать более емкий код, так как подразумевает использование все того же сокращения `border-radius`, но на этот раз **с несколькими значениями, разделенными пробелами**. Если указать четыре значения, то они будут применены к каждому из четырех углов **по часовой стрелке, начиная с верхнего левого**. Если указать меньше четырех значений, то они будут умножаться обычным способом, как принято в CSS, — аналогично тому, как это происходит со свойством `border-width`. Три значения подразумевают, что четвертое совпадает со вторым. Два значения подразумевают, что третье совпадает с первым. На рис. 3.7 вы найдете визуальное объяснение того, как это работает. Мы можем даже указать **разные горизонтальные и вертикальные радиусы для всех четырех углов**, перечислив 1–4 значения до слеша и 1–4 других значения

после него. Обратите внимание, что эти последовательности раскрываются в полные наборы из четырех значений по отдельности. Например, значение `border-radius`, равное `10px / 5px 20px`, эквивалентно `10px 10px 10px 10px / 5px 20px 5px 20px`.

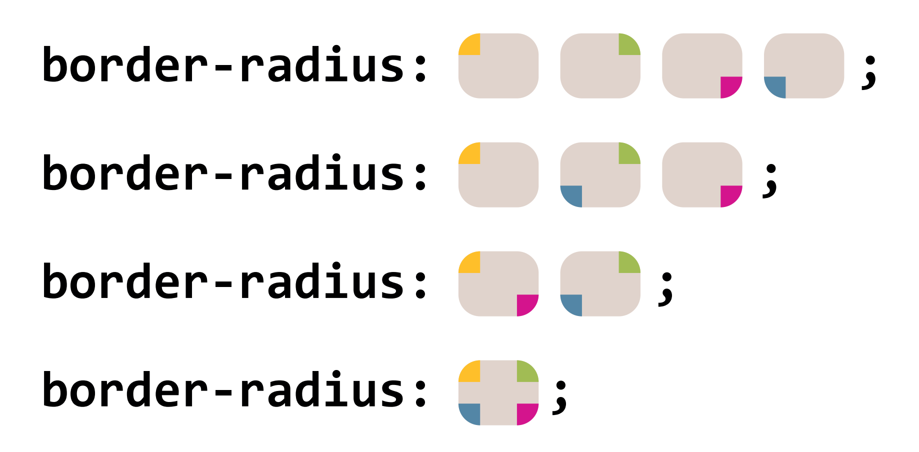


Рис. 3.7. Скругление, угол которого определяется четырьмя, тремя, двумя или одним значением свойства `border-radius` (значения разделяются пробелом). Обратите внимание, что для эллиптических радиусов можно указывать до четырех аргументов до и после слеша, и привязка к углам будет точно такой же (отдельно для горизонтальных радиусов до слеша и вертикальных радиусов после него)

Вооружившись этими знаниями, давайте снова вернемся к проблеме половины эллипса. Можно ли указать такие значения `border-radius`, чтобы в результате была сгенерирована требуемая фигура? Мы не узнаем, пока не попробуем. Начнем с того, что запишем некоторые результаты наблюдений:

- фигура **симметрична по горизонтали**, что означает, что **верхний левый и верхний правый радиусы должны быть одинаковыми**; точно так же должны совпадать **нижний левый и нижний правый радиусы**;
- наверху нет прямого горизонтального края (то есть вся верхняя граница фигуры изогнута), что означает, что **верхний левый и верхний правый радиусы в сумме должны давать 100% ширины фигуры**;
- исходя из предыдущих двух наблюдений, можно заключить, что горизонтальные радиусы, как левый, так и правый, должны составлять 50%;
- по вертикали **скругление двух верхних углов распространяется на всю высоту элемента,** а **у нижних углов скругление отсутствует**. Следовательно, разумными значениями для вертикальной части `border-radius` кажутся `100% 100% 0 0`;

- так как вертикальное скругление нижних углов равно нулю, их горизонтальное скругление не играет никакой роли, поскольку результат всегда будет нулевым (вы можете вообразить угол с нулевым вертикальным скруглением и положительным горизонтальным? Вот-вот, разработчики спецификации тоже не могли).

Сложив все это вместе, нетрудно сформулировать CSS-код для гибкой половины эллипса с рис. 3.6:

```
border-radius: 50% / 100% 100% 0 0;
```

Ничуть не сложнее сообразить, какими должны быть значения для создания половинки эллипса, разрезанного по вертикальной оси, как показанная на рис. 3.8:

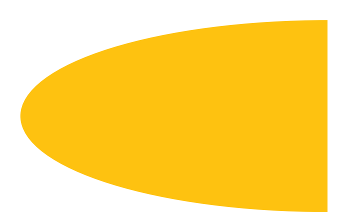


Рис. 3.8. Половина эллипса, разрезанного вдоль вертикальной оси

```
border-radius: 100% 0 0 100% / 50%;
```

В качестве упражнения попробуйте написать CSS-код для второй половинки эллипса.

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**half-ellipse**

Четвертинки эллипса

Аналогично примеру с половиной эллипса, когда ширина равна высоте, мы получаем четвертинку **круга**.

После создания целого эллипса и половины эллипса естественным образом возникает вопрос о четвертинке эллипса, например такой, как показана на рис. 3.9. Следуя той же цепочке размышлений, можно заметить, что для создания четвертинки эллипса необходимо **для одного из углов указать 100% радиус как по горизонтали, так и по вертикали, а остальные определить безо всякого скругления**. Поскольку процентное значение должно быть одинаковым и для горизонтальных, и для вертикальных радиусов всех четырех углов, запись со слешем не требуется. Код будет выглядеть так:



Рис. 3.9. Четвертинка эллипса

```
border-radius: 100% 0 0 0;
```

К сожалению, на случай, если вы теперь размышляете о других долях эллипса, которые можно было бы создавать с помощью `border-radius` (например, можно ли сконструировать одну восьмую эллипса? или одну треть?), должна вас огорчить: `border-radius` не поддерживает значения, которые бы позволили реализовать это.



Рис. 3.10. Simurai мастерски применил возможности `border-radius` для создания всевозможных типов фигур для своих кнопок-конфеток (**BonBon buttons** — http://simurai.com/archive/buttons)

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**quarter-ellipse**

СВЯЗАННЫЕ СПЕЦИФИКАЦИИ

CSS Backgrounds & Borders: http://w3.org/TR/css-backgrounds

10 Параллелограммы

ПРЕДВАРИТЕЛЬНЫЕ ТРЕБОВАНИЯ

Базовое знание трансформаций CSS

Проблема

Параллелограммы — это расширенная версия прямоугольников: их стороны параллельны, но углы не обязательно прямые (рис. 3.11). В визуальном дизайне они часто используются для придания оформлению динамичности и передачи ощущения движения (рис. 3.12).



Рис. 3.11. Параллелограмм

Давайте попробуем создать ссылку, оформленную с помощью CSS в стиле скошенной кнопки. Нашей отправной точкой будет обычная плоская кнопка с очень простым оформлением, такая, например, как на рис. 3.13. Форму скошенного прямоугольника мы придадим ей с помощью трансформации `skew()`, вот так:

```
transform: skewX(-45deg);
```

Однако в результате этого содержимое кнопки также исказилось, стало некрасивым и нечитаемым (рис. 3.14). **Существует ли способ создавать скошенные контейнеры так, чтобы их содержимое при этом не перекашивалось?**

Решение с вложенными элементами

Для получения желаемого результата мы могли бы **применить к содержимому трансформацию, противоположную `skew()`, которая отменит внешнюю трансформацию**. К сожалению, это означает, что нам придется использовать дополнительный элемент HTML в качестве обертки содержимого, например `div`:

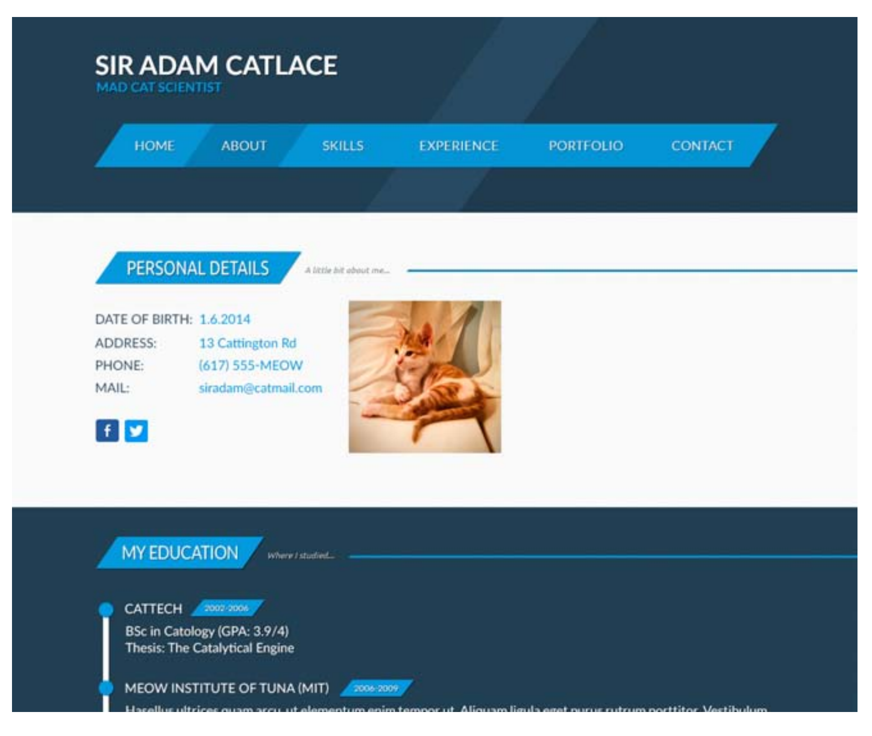


Рис. 3.12. Параллелограммы в веб-дизайне (автор дизайна — Мартина Питакова (Martina Pitakova))

CLICK ME

Рис. 3.13. Наша кнопка до применения каких-либо трансформаций



Рис. 3.14. Наша скошенная кнопка, текст на которой трудно прочитать

HTML

```
<a href="#yolo" class="button">
    <div>Click me</div>
</a>
```

```
.button { transform: skewX(-45deg); }
.button > div { transform: skewX(45deg); }
```

Как видно на рис. 3.15, этот подход работает, но требует дополнительного элемента HTML. Однако не следует беспокоиться, если изменение разметки для вас неприемлемо или же если вы действительно стремитесь к сохранению чистоты разметки, — существует также решение на чистом CSS.

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**parallelograms**

Решение с псевдоэлементом

Еще одна идея — **создать псевдоэлемент, к которому будут применены все стили** (фоны, рамки и т. п.), а затем **трансформировать его**. Так как наше содержимое не принадлежит псевдоэлементу, трансформация на него распространяться не будет. Попробуем применить эту технику для стилизации ссылки так, как в предыдущем разделе.



Если вы применяете этот эффект к элементу, который по умолчанию является строковым (inline), то не забудьте установить для него другое значение свойства display, например inline-block или block, иначе **трансформация применена не будет**. То же самое касается и внутреннего элемента.

Нам нужно, чтобы поле нашего псевдоэлемента оставалось гибким и автоматически наследовало габаритные размеры своего предка, даже когда они определяются содержимым. Самый простой способ добиться этого — применить `position: relative` к предку, а к сгенерированному содержимому — `position: absolute` и сделать все смещения нулевыми, чтобы оно растягивалось по горизонтали и по вертикали до размеров своего предка. Вот как будет выглядеть требуемый код:

```
.button {
    position: relative;
    /* цвет текста, забивки и т. п. */
}
.button::before {
    content: '';
    position: absolute;
    top: 0; right: 0; bottom: 0; left: 0;
}
```

CLICK ME

Рис. 3.15. Конечный результат

Пока что сгенерированное поле отображается над содержимым, заслоняя его, и как только мы определяем для него какой-то фон, содержимое становится невидимым (рис. 3.16). Чтобы исправить это, применим `z-index: -1` к псевдоэлементу, для того чтобы он переместился ниже своего предка.

Рис. 3.16. Наш псевдоэлемент в настоящее время находится выше своего содержимого, поэтому применение к нему `background: #58a` приводит к тому, что увидеть содержимое становится невозможно

После этого наконец-то можно применить к псевдоэлементу требуемые трансформации и наслаждаться результатом. Итоговый вариант кода показан далее; он обеспечивает тот же самый визуальный результат, что и предыдущая техника:

```
.button {
    position: relative;
    /* цвет текста, забивки и т. п. */
```

```
}
.button::before {
    content: ''; /* чтобы сгенерировать поле */
    position: absolute;
    top: 0; right: 0; bottom: 0; left: 0;
    z-index: -1;
    background: #58a;
    transform: skew(45deg);
}
```

Эти техники полезны не только при применении трансформации `skew()`. Их можно использовать с **любыми другими трансформациями, для того чтобы менять форму элемента, не воздействуя на его содержимое**. Например, применив вариацию данной техники с трансформацией `rotate()` к квадратному элементу, вы с легкостью создадите ромб.

Кроме того, идея использовать псевдоэлементы и позиционирование для генерации поля, которое затем стилизуется и помещается под своего предка, может пригодиться во множестве других ситуаций для создания самых разных типов эффектов, например:

- данную технику часто использовали в качестве обходного пути при добавлении нескольких фонов в IE8 (автор техники — Николас Галлахер (Nicolas Gallagher); http://nicolasgallagher.com/multiple-backgrounds-and-borders-with-css2);
- это может быть еще одним решением для создания эффектов, подобных описанному в **секрете «Внутреннее скругление»**. Догадаетесь почему?
- с помощью этой техники можно независимо применить свойства, подобные `opacity`, к «фону», что также впервые воплотил Николас Галлахер (http://nicolasgallagher.com/css-background-image-hacks);
- она предоставляет более гибкий способ имитации нескольких рамок на случай, если вы не можете применить техники из **секрета «Несколько рамок»**. Например, когда вам требуется несколько пунктирных рамок или несколько рамок с пустым пространством, заполненным прозрачными пикселами, между ними.

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**parallelograms-pseudo**

СВЯЗАННЫЕ СПЕЦИФИКАЦИИ

CSS Transforms: http://w3.org/TR/css-transforms

11 Изображения в форме ромба

ПРЕДВАРИТЕЛЬНЫЕ ТРЕБОВАНИЯ

Трансформации CSS, **секрет «Параллелограммы»**

Проблема

Обрезка изображений до ромбовидной формы — распространенный прием в визуальном дизайне, но реализовать его на CSS далеко не просто. На самом деле до недавнего времени это было практически невозможно. Поэтому для



Рис. 3.17. После обновления дизайна в 2013 году портреты авторов в профиле на веб-сайте 24ways.org теперь отображаются в форме ромба. Это сделано с помощью техники из данного раздела

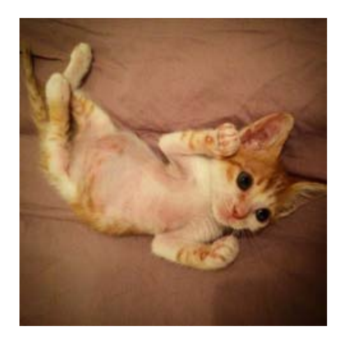


Рис. 3.18. Наше исходное изображение, которое мы собираемся обрезать по форме ромба

воплощения своих задумок дизайнерам приходилось сперва обрезать требуемые изображения в графическом редакторе. Разумеется, не нужно и говорить, что такой вариант применения эффекта означает огромные сложности в сопровождении веб-сайта и гарантированную неразбериху в будущем, если кто-то пожелает изменить стилизацию изображений.

Определенно, сегодня у нас уже должен быть способ получше. В действительности таких способов целых **два**!

Решение на основе трансформации

Основная идея та же, что и в первом решении из предыдущего секрета (см. **секрет «Параллелограммы»** выше), — нам необходимо обернуть наше изображение в `<div>`, а затем применить к нему противоположную трансформацию `rotate()`:



Рис. 3.19. Противоположных трансформаций `rotate()` недостаточно для достижения нужного эффекта (блок `div` с названием `.picture` обозначен пунктирным контуром)

HTML

```
<div class="picture">
    <img src="adam-catlace.jpg" alt="…" />
</div>
```

```
.picture {
    width: 400px;
    transform: rotate(45deg);
    overflow: hidden;
}
.picture > img {
    max-width: 100%;
    transform: rotate(-45deg);
}
```

Однако, как вы видите на рис. 3.19, у нас не получилось с наскока добиться требуемой стилизации. Конечно, если вы планировали обрезать изображение по форме восьмиугольника, то можете сказать, что работа сделана, и заняться чем-то еще. Но для того чтобы обрезать картинку по форме ромба, придется еще попотеть.

Главная проблема кроется в объявлении `max-width: 100%`. `100%` относится к стороне нашего контейнера `.picture`. Однако мы хотим, чтобы **ширина итогового изображения была равна диагонали исходного, а не его стороне**. Вы уже догадались, что нам опять требуется помощь теоремы Пифагора (если вам необходимо освежить ее в памяти, то объяснение вы найдете в **секрете «Диагональные полосы»**). Как гласит теорема, диагональ квадрата равна его стороне, умноженной на $\sqrt{2} \approx 1{,}414213562$. Следовательно, имеет смысл задать значение `max-width`, равное $\sqrt{2} \times 100\% \approx 141{,}4213562\%$ или, округляя, `142%`, так как мы ни в коем случае не хотим, чтобы изображение уменьшилось (а если оно окажется **чуть больше, то все в порядке**, поскольку мы все равно его обрезаем).

В действительности еще лучше увеличивать изображение посредством трансформации `scale()`, и тому есть две причины:

- мы хотим, чтобы в ситуации, когда трансформации CSS не поддерживаются, размер изображения оставался равным 100%;
- при увеличении изображения посредством трансформации `scale()` оно масштабируется от центра (если не указано иное значение `transform-origin`). Если вы будете увеличивать его путем изменения значения свойства `width`, то оно будет масштабироваться от верхнего левого угла и для того, чтобы переместить его, нам понадобится использовать отрицательные значения полей.

Складывая все вместе, получаем такой финальный вариант кода:

```
.picture {
    width: 400px;
    transform: rotate(45deg);
    overflow: hidden;
}
.picture > img {
    max-width: 100%;
    transform: rotate(-45deg) scale(1.42);
}
```

Как видно на рис. 3.20, это наконец-то дает нам желаемый результат.



Рис. 3.20. Наше готовое обрезанное изображение

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**diamond-images**

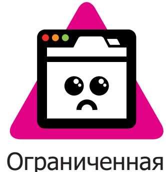


Рис. 3.21. Решение, основанное на трансформациях, некрасиво ломается, когда мы пытаемся применить его с неквадратными изображениями



Рис. 3.22. Метод на основе `clip-path` отлично подходит для неквадратных изображений

Решение с обтравочным контуром



Предыдущее решение работает, но по своей природе это грязный трюк. Он требует дополнительного элемента HTML, а значит, это беспорядочное, запутанное и хрупкое решение: если нам придется иметь дело с неквадратными изображениями, результат будет печальным (рис. 3.21).

В действительности существует намного лучший способ достичь желаемого результата. Основная идея заключается в использовании свойства `clip-path` — еще одной возможности, позаимствованной из SVG. Это свойство теперь можно применять и к HTML-содержимому (по крайней мере, в поддерживающих браузерах), причем с использованием приятного и читабельного синтаксиса, в отличие от эквивалента в SVG, печально известного своим умением доводить людей до бешенства. У него есть лишь один недостаток (на момент написания этой главы) — ограниченная поддержка браузерами. Однако данное решение изящно откатывается до упрощенной визуализации (без обрезки), так что это достойная кандидатура для рассмотрения.

Скорее всего, вы уже знакомы с обтравочными контурами благодаря приложениям для редактирования изображений, таким как Adobe Photoshop. Обтравочные контуры позволяют обрезать элемент до любой формы, какую вы только пожелаете. В данном случае мы собираемся использовать фигуру `polygon()`. Мы будем определять ромб, но в целом эта фигура позволяет задать любой многоугольник последовательностью точек, разделенных запятыми. Можно даже использовать проценты — итоговые значения будут вычисляться относительно габаритных размеров элемента. Код очень простой:

```
clip-path: polygon(50% 0, 100% 50%, 50% 100%,
                   0 50%);
```

Верите или нет, но это все! Результат идентичен показанному на рис. 3.20, но вместо двух элементов HTML и восьми строк запутанного кода CSS мы достигли желаемого с помощью всего лишь одной простой строки.

Но этим чудесные способности `clip-path` не ограничиваются. Это свойство поддерживает даже анимацию — при условии, что мы анимируем переход между двумя одинаковыми функциями фигур (в нашем случае `polygon()`) с одинаковым количеством точек. Таким образом, если мы хотим плавно раскрывать полное изображение при наведении указателя мыши, это можно реализовать таким способом:

```
img {
    clip-path: polygon(50% 0, 100% 50%,
                       50% 100%, 0 50%);
    transition: 1s clip-path;
}

img:hover {
    clip-path: polygon(0 0, 100% 0,
                       100% 100%, 0 100%);
}
```

Кроме того, этот метод прекрасно приспосабливается к неквадратным изображениям, что подтверждает рис. 3.22. Ах, радости современного CSS...

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**diamond-clip**

СВЯЗАННЫЕ СПЕЦИФИКАЦИИ

CSS Transforms: http://w3.org/TR/css-transforms

CSS Masking: http://w3.org/TR/css-masking

CSS Transitions: http://w3.org/TR/css-transitions

12 Срезанные углы

ПРЕДВАРИТЕЛЬНЫЕ ТРЕБОВАНИЯ

Градиенты CSS, знание свойства `background-size`, секрет «Фоны в полоску»

Проблема

Срезание углов — это не только быстрый способ достичь цели, но и популярный вариант стилизации как в печатном дизайне, так и в веб-дизайне. Чаще всего он подразумевает обрезание одного или нескольких уголков контейнера под углом 45°. В последнее время, в связи с тем, что скевоморфизм начал сдавать позиции плоскому дизайну, этот эффект пользуется особенной популярностью. Когда углы срезаются только с одной стороны и каждый из них занимает 50% высоты элемента, это создает фигуру в форме стрелки, что также часто используется в оформлении кнопок и элементов навигации типа «хлебные крошки» (рис. 3.23).



Рис. 3.23. Кнопка со срезанными углами выглядит как стрелка, что подчеркивает ее предназначение

Однако в CSS все еще недостаточно инструментов для создания этого эффекта с помощью простых и понятных однострочных решений. Из-за этого многие разработчики склоняются к использованию фоновых изображений: либо закрывают срезанные углы треугольниками (на одноцветном фоне), либо создают весь фон с помощью одного или нескольких изображений, где углы уже срезаны.

Очевидно, что такие методы совершенно негибкие, они сложны в сопровождении и увеличивают время ожидания вследствие дополнительных HTTP-запросов и общего размера файлов веб-сайта.



Рис. 3.24. Пример веб-сайта, где срезанный угол (нижний левый у полупрозрачного поля *Find & Book*) отлично вписывается в дизайн

Решение

Одно возможное решение предлагают нам всемогущие градиенты CSS. Предположим, что нам требуется только **один срезанный угол**, скажем, нижний правый. Трюк в том, чтобы воспользоваться способностью градиентов принимать направление угла (например, `45deg`) и позиции границ перехода цвета в абсолютных значениях, которые **не меняются при изменении габаритных размеров элемента**, которому принадлежит фон.

Из вышесказанного следует, что нам будет достаточно одного линейного градиента. Мы добавим прозрачную границу перехода цвета для создания срезанного угла и еще одну границу перехода цвета в той же позиции, но уже с цветом, соответствующим фону. Код CSS будет следующим (для угла размером `15px`):

```
background: #58a;
background:
    linear-gradient(-45deg, transparent 15px, #58a 0);
```

Просто, не так ли? Результат вы видите на рис. 3.25. Технически первое объявление нам даже не требуется. Мы добавили его только в качестве **обходного пути**: если градиенты CSS не поддерживаются, то второе объявление будет проигнорировано, так что мы **как минимум** получим фон сплошного цвета.

Эй, сосредоточься!
На углы смотри,
а не текст читай!
Текст только для примера!

Рис. 3.25. Элемент со срезанным нижним правым углом, созданный с помощью простого градиента CSS

Теперь предположим, что нам требуются **два срезанных угла**, скажем, оба нижних. Это невозможно реализовать с помощью одного градиента, так что нам понадобятся два. Первой мыслью может быть нечто подобное:

СОВЕТ

Мы использовали **разные цвета** (#58a **и** #655) **для упрощения отладки**. На практике оба градиента будут одного и того же цвета.

```
background: #58a;
background:
    linear-gradient(-45deg, transparent 15px,
                    #58a 0),
    linear-gradient(45deg, transparent 15px,
                    #655 0);
```

Однако, как можно видеть на рис. 3.26, это не работает. По умолчанию оба градиента занимают всю площадь элемента, так что они **заслоняют друг друга**. Мы должны уменьшить их, ограничив каждый из них **половиной элемента** с помощью `background-size`:

Эй, сосредоточься!
На углы смотри,
а не текст читай!
Текст только для примера!

Рис. 3.26. Провалившаяся попытка применить эффект срезанного угла к обоим нижним углам

```
background: #58a;
background:
    linear-gradient(-45deg, transparent 15px,
                    #58a 0)
        right,
    linear-gradient(45deg, transparent 15px,
                    #655 0)
        left;
background-size: 50% 100%;
```

Эй, сосредоточься!
На углы смотри,
а не текст читай!
Текст только для примера!

Рис. 3.27. Помощи `background-size` недостаточно

Результат вы можете видеть на рис. 3.27. Несмотря на то что мы применили `background-size`, градиенты все равно перекрывают друг друга. Причина в том, что мы забыли выключить `background-repeat`, поэтому **каждый из фонов повторяется дважды**. Следовательно, один из фонов все так же заслоняет другой, но на этот раз за счет повторения. Новая версия кода выглядит так:

```
background: #58a;
background:
    linear-gradient(-45deg, transparent 15px,
        #58a 0) right,
    linear-gradient(45deg, transparent 15px,
        #655 0) left;
background-size: 50% 100%;
background-repeat: no-repeat;
```

Результат вы можете увидеть на рис. 3.28 и убедиться, что он — наконец-то! — работает! Вы наверняка уже догадались, как применить этот эффект ко всем четырем углам. Вам потребуются четыре градиента и код, подобный следующему:

```
background: #58a;
background:
    linear-gradient(135deg, transparent 15px,
                   #58a 0)
        top left,
    linear-gradient(-135deg, transparent 15px,
                   #655 0)
        top right,
    linear-gradient(-45deg, transparent 15px,
                   #58a 0)
        bottom right,
    linear-gradient(45deg, transparent 15px,
                   #655 0)
        bottom left;
background-size: 50% 50%;
background-repeat: no-repeat;
```

Рис. 3.28. Теперь оба нижних угла, левый и правый, успешно срезаются

Рис. 3.29. Эффект, примененный ко всем четырем углам посредством четырех градиентов

Результат показан на рис. 3.29. Но проблема предыдущего кода в том, что он трудно поддается сопровождению. Он требует внести **пять правок для изменения фонового цвета** и **четыре для изменения величины угла**. Примесь, созданная с помощью препроцессора, могла бы сократить количество повторений. Вот как этот код будет выглядеть в SCSS:

SCSS

```
@mixin beveled-corners($bg,
        $tl:0, $tr:$tl, $br:$tl, $bl:$tr) {
    background: $bg;
    background:
        linear-gradient(135deg, transparent $tl, $bg 0)
            top left,
        linear-gradient(225deg, transparent $tr, $bg 0)
            top right,
        linear-gradient(-45deg, transparent $br, $bg 0)
            bottom right,
        linear-gradient(45deg, transparent $bl, $bg 0)
            bottom left;
    background-size: 50% 50%;
    background-repeat: no-repeat;
}
```

Затем, когда необходимо, его можно будет вызывать, как показано далее, с 2–5 аргументами:

SCSS

```
@include beveled-corners(#58a, 15px, 5px);
```

В этом примере мы получим элемент, у которого верхний левый и нижний правый углы срезаны на `15px`, а верхний правый и нижний левый — на `5px`, аналогично тому, как работает `border-radius`, когда мы указываем меньше четырех значений. Это возможно благодаря тому, что мы в нашей примеси SCSS также позаботились о значениях по умолчанию для аргументов, — и да, эти значения по умолчанию могут ссылаться и на другие аргументы тоже.

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**bevel-corners-gradients**

Искривленные срезанные углы

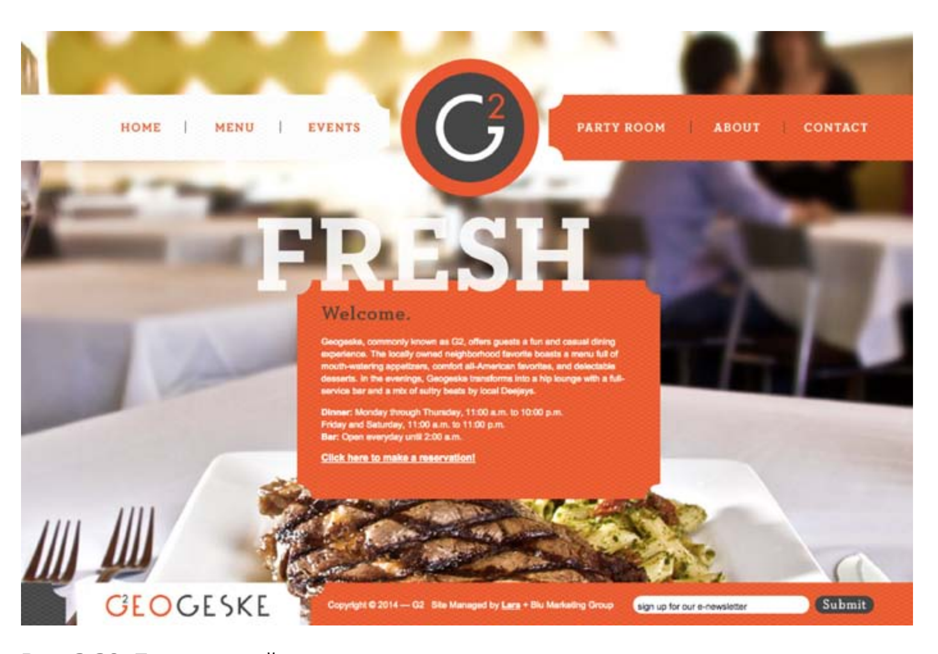


Рис. 3.30. Превосходный пример использования искривленных срезанных углов на веб-сайте http://g2geogeske.com; дизайнер сделал их центральным элементом оформления: они присутствуют в навигации, в содержимом и даже в нижнем колонтитуле 

Вариация метода с градиентами позволяет создавать искривленные срезанные углы — эффект, который многие называют «внутренним радиусом рамки», так как он выглядит словно инвертированная версия скругленных углов. Единственное отличие заключается в использовании радиальных градиентов вместо линейных:

```
background: #58a;
background:
    radial-gradient(circle at top left,
        transparent 15px, #58a 0) top left,
    radial-gradient(circle at top right,
        transparent 15px, #58a 0) top right,
    radial-gradient(circle at bottom right,
        transparent 15px, #58a 0) bottom right,
    radial-gradient(circle at bottom left,
        transparent 15px, #58a 0) bottom left;
background-size: 50% 50%;
background-repeat: no-repeat;
```

Результат показан на рис. 3.31. Так же как и в предыдущей технике, размером угла можно управлять посредством позиций границ перехода цвета, а примесь способна и этот код сделать более пригодным для дальнейшего сопровождения.

Эй, сосредоточься!
На углы смотри,
а не текст читай!
Текст только для примера!

Рис. 3.31. Искривленные срезанные углы, сделанные с помощью радиальных градиентов

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**scoop-corners**

Решение со строковым SVG и border-image

Хотя решение, основанное на градиентах, работает, у него есть несколько недостатков:

- код очень **длинный и полон повторений**. В самом распространенном случае, когда нам требуется обрезать все четыре угла на одинаковую величину, изменение этой величины влечет за собой четыре правки в коде. Аналогично, для изменения фонового цвета также необходимы четыре правки, а если учитывать резервное решение, то все пять;
- анимировать изменение величины срезанного угла невероятно сложно, а в некоторых браузерах вообще невозможно.

К счастью, в зависимости от желаемого результата мы можем воспользоваться еще парой методов. Один из них подразумевает объединение `border-image` со строковым SVG-кодом, в котором и генерируются углы. Зная, как работает `border-image` (если вам необходимо освежить эти знания в памяти, подсказку вы найдете на рис. 2.58), можете ли вы уже представить, как должен выглядеть требуемый SVG-код?

Так как габаритные размеры для нас неважны (`border-image` позаботится о масштабировании, а SVG-рисунки идеально масштабируются вне зависимости от

габаритов — будь благословенна векторная графика!), все размеры можно приравнять к единице, для того чтобы оперировать более удобными и короткими значениями. Величина срезанного угла будет равна единице, и прямые стороны также будут равны единице. Результат (увеличенный для удобства восприятия) будет выглядеть как на рис. 3.32. Код, необходимый для этого, показан далее:

```
border: 15px solid transparent;
border-image: 1 url('data:image/svg+xml,\
    <svg xmlns="http://www.w3.org/2000/svg"
        width="3" height="3" fill="%2358a">\
    <polygon points="0,1 1,0 2,0 3,1 3,2 2,3 1,3 0,2"/>\
    </svg>');
```

Обратите внимание, что размер шага нарезки равен `1`. Это не означает 1 пиксел; фактический размер определяется системой координат SVG-файла (потому-то у нас и отсутствуют единицы измерения). Если бы мы использовали проценты, то нам пришлось бы аппроксимировать 1/3 изображения дробным значением, вроде `33.34%`. Прибегать к приблизительным значениям всегда рискованно, так как в разных браузерах значения могут округляться с разной степенью точности. А придерживаясь единиц изменения системы координат SVG-файла, мы избавляем себя от головной боли, сопутствующей всем этим округлениям.

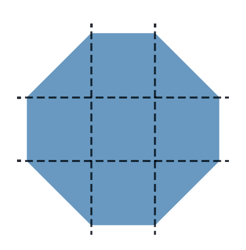


Рис. 3.32. Наше изображение для рамки, сделанное с помощью SVG, и соответствующая нарезка

Результат показан на рис. 3.33. Как вы видите, срезанные углы присутствуют, но фона нет. Эту проблему можно решить двумя способами: либо определить фон, либо добавить ключевое слово `fill` к объявлению `border-image`, **чтобы центральный элемент нарезки не отбрасывался**. В нашем примере мы лучше определим отдельный фон, так как это определение **будет также служить обходным путем** для браузеров, не поддерживающих данное решение.



Рис. 3.33. Использование SVG-кода со свойством `border-image`

Помимо этого, вы, вероятно, заметили, что теперь **срезанные углы меньше, чем при использовании предыдущей техники**, и это может поставить в тупик. Мы ведь задали ширину рамки, равную `15px`! Причина в том, что в решении с градиентом эти 15 пикселов отсчитывались вдоль *градиентной линии,* которая перпендикулярна направлению градиента. Однако ширина рамки измеряется не по диагонали, а по горизонтали/вертикали. Чувствуете, к чему я веду? Да-да, снова вездесущая теорема Пифагора, которую мы активно использовали в **секрете «Фоны в полоску»**. Схема на рис. 3.34

должна прояснить ситуацию. Короче говоря, для того, чтобы достичь того же визуального результата, нам необходима ширина рамки, в $\sqrt{2}$ раз превышающая размер, который мы бы использовали в методе с градиентом. В данном случае это будет $15 \times \sqrt{2} \approx 21{,}213203436$ пиксела, что разумнее всего аппроксимировать до `20px`, если только перед нами не стоит задача как можно точнее приблизить размер диагонали к заветным `15px`:

```
border: 20px solid transparent;
border-image: 1 url('data:image/svg+xml,\
    <svg xmlns="http://www.w3.org/2000/svg"
        width="3" height="3" fill="%2358a">\
    <polygon points="0,1 1,0 2,0 3,1 3,2 2,3 1,3
                0,2"/>\
    </svg>');
background: #58a;
```

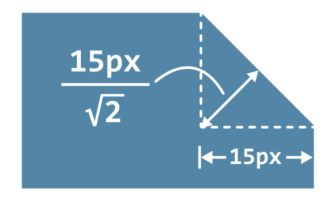


Рис. 3.34. Значение `border-width`, равное `15px`, создает угол размером $15/\sqrt{2} \approx 10{,}606601718$ (если измерять по диагонали), поэтому наши срезанные углы кажутся меньше

Эй, сосредоточься!
На углы смотри,
а не текст читай!
Текст только для примера!

Рис. 3.35. Куда делись наши миленькие углы?

Однако, как можно видеть на рис. 3.35, результат не совсем тот, которого мы ожидали. Куда делись наши кропотливо срезанные углы? Не бойся, юный падаван, углы все так же на месте. Вы сразу же поймете, что произошло, если установите другой фоновый цвет, например `#655`.

Как демонстрирует рис. 3.36, причина, почему наши углы исчезли, кроется в фоне: тот фон, который мы выше определили, попросту заслоняет их. Все, что нам нужно сделать для устранения этого неудобства, — с помощью `background-clip` запретить фону подлезать под область рамки:

Эй, сосредоточься!
На углы смотри,
а не текст читай!
Текст только для примера!

Рис. 3.36. Установив для свойства `background` другой цвет, мы решаем загадку… исчезнувших углов

```
border: 20px solid transparent;
border-image: 1 url('data:image/svg+xml,\
    <svg xmlns="http://www.w3.org/2000/svg"\
        width="3" height="3" fill="%2358a">\
    <polygon points="0,1 1,0 2,0 3,1 3,2 2,3 1,3
                0,2"/>\
    </svg>');
background: #58a;
background-clip: padding-box;
```

Теперь проблема решена, и наше поле выглядит точно так же, как на рис. 3.29. К тому же на этот

Эй, сосредоточься!
На углы смотри,
а не текст читай!
Текст только для примера!

Рис. 3.37. Элемент со срезанными углами и радиальным градиентом на фоне

раз мы можем с легкостью **изменить размер углов, внеся всего лишь одну правку**: просто скорректируем ширину рамки. **Мы можем даже анимировать этот эффект**, потому что `border-width` поддерживает анимацию! А для смены фона требуется теперь **две правки вместо пяти**. Кроме того, так как наш фон не зависит от эффекта, накладываемого на углы, мы можем определить для него градиент или любой другой узор, при условии, что по краям цвет все так же будет равен `#58a`. Например, на рис. 3.37 мы используем радиальный градиент от цвета `hsla(0,0%,100%,.2)` до `transparent`.

Осталось решить лишь одну небольшую проблему. Если `border-image` не поддерживается, то резервное решение не ограничится отсутствием срезанных углов. Из-за того что фон обрезан, пространство между краем поля и его содержимым уменьшится. Для того чтобы исправить это, необходимо для рамки определить тот же цвет, который мы используем для фона:

```
border: 20px solid #58a;
border-image: 1 url('data:image/svg+xml,\
    <svg xmlns="http://www.w3.org/2000/svg"\
        width="3" height="3" fill="%2358a">\
    <polygon points="0,1 1,0 2,0 3,1 3,2 2,3 1,3 0,2"/>\
    </svg>');
background: #58a;
background-clip: padding-box;
```

В браузерах, где наше определение `border-image` поддерживается, этот цвет будет проигнорирован, но там, где `border-image` не работает, дополнительный цвет рамки обеспечит более изящное резервное решение, которое выглядит как на рис. 3.35. Единственный его недостаток — **увеличение количества правок**, необходимых для изменения фонового цвета, до трех.

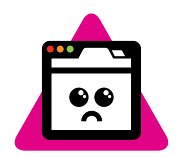
ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**bevel-corners**


Благодарности

Спасибо Мартину Сали (Martijn Saly, http://twitter.com/martijnsaly) за первоначальную идею использовать `border-image` и строковый SVG-код в качестве решения для эффекта срезанных углов. Ссылка была опубликовала в твите от 5 января 2015 года: http://twitter.com/martijnsaly/status/552152520114855936.

Решение с обтравочным контуром


Ограниченная поддержка

Хотя решение с `border-image` очень компактное и хорошо соответствует принципам DRY, оно накладывает определенные ограничения. Например, наш фон все так же должен быть либо целиком, либо хотя бы вдоль кромок заполнен сплошным цветом. А что, если мы хотим использовать другой тип фона, например текстуру, узор или линейный градиент?

Существует другой способ, не имеющий подобных ограничений, хотя, конечно, определенные ограничения его применения есть. Помните свойство `clip-path` из **секрета «Изображения в форме ромба»**? Обтравочные контуры CSS обладают поразительным свойством: они позволяют смешивать процентные значения (с помощью которых мы указываем габаритные размеры элемента) с абсолютными, обеспечивая невероятную гибкость.

Например, код обтравочного контура, обрезающего элемент до формы прямоугольника со скошенными углами размером `20px` (если измерять по горизонтали), выглядит так:

```
background: #58a;
clip-path: polygon(
    20px 0, calc(100% - 20px) 0, 100% 20px,
    100% calc(100% - 20px), calc(100% - 20px) 100%,
    20px 100%, 0 calc(100% - 20px), 0 20px
);
```

Несмотря на краткость, в этом фрагменте кода принципы DRY не соблюдены, и это становится одной из самых больших проблем, если вы не используете препроцессор. В действительности этот код — лучшая иллюстрация принципа WET из всех решений на чистом CSS, представленных в этой книге, ведь для изменения размера угла здесь требуется внести целых восемь (!) правок. С другой стороны, фон можно поменять с помощью только одной правки, так что у нас есть хотя бы это.



Рис. 3.38. Изображение, для стилизации которого использованы срезанные углы; реализация посредством `clip-path`

Одно из преимуществ данного подхода — то, что мы можем использовать абсолютно любой фон или даже обрезать подменные элементы, такие как изображения. На рис. 3.38 показано изображение, стилизованное с использованием срезанных углов. Ни один из предыдущих методов не позволяет добиться такого эффекта. Помимо этого, свойство `clip-path` поддерживает анимацию, и мы можем анимировать не только изменение размера угла, но и переходы

между разными фигурами. Все, что для этого нужно, — использовать другой обтравочный контур.

Помимо многословности и ограниченной поддержки браузерами, недостатком этого решения является то, что, **если мы не позаботимся о достаточно широкой забивке, текст также будет обрезан**, так как при обрезке элемента его составляющие никак не учитываются. В противоположность этому метод с градиентом позволяет тексту просто выходить за границы обрезанных углов (ведь они всего лишь часть фона), а метод с `border-image` работает так же, как обычные рамки, — переносит текст на новую строку.

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**bevel-corners-clipped**

СВЯЗАННЫЕ СПЕЦИФИКАЦИИ

CSS Backgrounds & Borders: http://w3.org/TR/css-backgrounds

CSS Image Values: http://w3.org/TR/css-images

CSS Transforms: http://w3.org/TR/css-transforms

CSS Masking: http://w3.org/TR/css-masking

CSS Transitions: http://w3.org/TR/css-transitions

CSS Backgrounds & Borders Level 4: http://dev.w3.org/csswg/css-backgrounds-4

БУДУЩЕЕ. СРЕЗАННЫЕ УГЛЫ

В будущем, для того чтобы воплотить эффект срезанных углов, нам не придется прибегать к помощи градиентов CSS, обрезки или SVG. Новое свойство `corner-shape`, входящее в состав **CSS Backgrounds & Borders Level 4** (http://dev.w3.org/csswg/cssbackgrounds-4/), спасет нас от этой головной боли. Оно будет использоваться для создания эффекта срезанных по разной форме углов в сочетании со свойством `border-radius`, которое необходимо для определения величины обрезки. Например, для описания срезанных углов размером 15px по всем сторонам изображения достаточно такого простого кода:

```
border-radius: 15px;
corner-shape: bevel;
```

13 Вкладки в форме трапеций

ПРЕДВАРИТЕЛЬНЫЕ ТРЕБОВАНИЯ

Базовые трехмерные трансформации, **секрет «Параллелограммы»**

Проблема

Трапеции — это еще более генерализованные фигуры, чем параллелограммы: у них только две параллельные стороны. Две оставшиеся могут быть наклонены под любым углом. Эти фигуры традиционно славятся **сложностью создания с помощью чистого CSS**, но при этом очень часто используются в веб-дизайне, особенно для оформления вкладок. Если разработчики не имитируют их посредством скрупулезно подготовленных фоновых изображений, то воссоздают с помощью прямоугольника с двумя треугольниками, сделанными из рамок, по бокам (рис. 3.39).

Trapezoid

Рис. 3.39. Имитация трапеции с помощью рамок псевдоэлементов (для наглядности псевдоэлементы обозначены более темным оттенком голубого)

Хотя эта техника экономит HTTP-запросы, которые могли бы понадобиться вследствие использования дополнительных изображений, и легко адаптируется к изменению размеров элементов, она все же далека от совершенства. Нам приходится создавать практически бесполезные псевдоэлементы, а кроме того, мы невероятно ограничены в вариантах стилизации. Например, попробуйте добавить рамку, фоновый узор или скругленные углы к такой вкладке.



Рис. 3.40. В Cloud9 (http://c9.io) все открытые документы отображаются на вкладках в форме трапеций

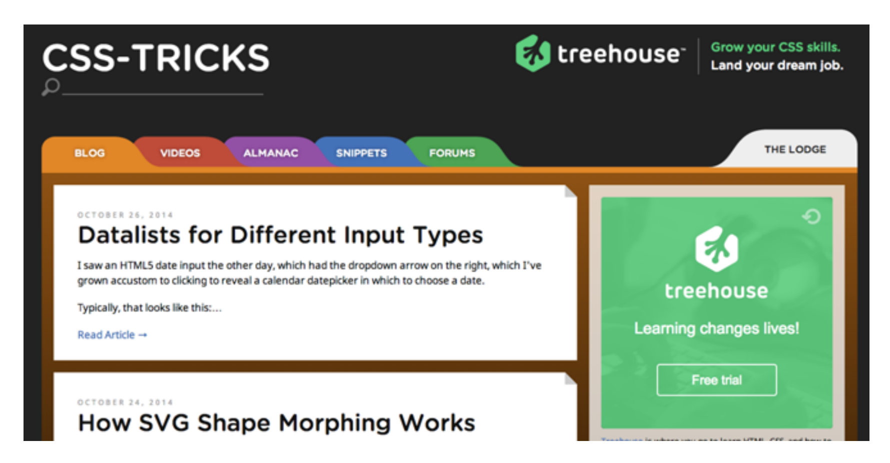


Рис. 3.41. В одном из предыдущих дизайнов веб-сайта https://css-tricks.com/ также фигурировали вкладки в форме трапеций, хотя они были скошены только с одной стороны

Так как все хорошо известные техники создания трапеций довольно запутанны и порождают хаотичный, сложный в поддержке код, большинство вкладок, которые мы встречаем в Сети, не имеют скошенных сторон, хотя реальные вкладки в приложениях чаще всего выглядят как раз как трапеции. Существует ли разумный и гибкий способ определять трапециевидные вкладки?

Решение

Если бы существовала комбинация двумерных трансформаций, позволяющая создавать трапециевидные фигуры, то мы могли бы просто применить вариацию решения из **секрета «Параллелограммы»** и покончить с этим. К сожалению, такой комбинации не существует.

Однако вообразите вращение прямоугольника в физическом, трехмерном мире. Чаще всего получившаяся фигура, благодаря перспективе, в двумерной

проекции выглядит именно как трапеция. И этот эффект можно имитировать в CSS с помощью трехмерного вращения:

```
transform: perspective(.5em) rotateX(5deg);
```

На рис. 3.42 видно, что в результате получается трапеция. Разумеется, так как мы применили трехмерную трансформацию ко всему элементу, искажение распространилось и на текст тоже. **Трехмерные трансформации невозможно «отменять» внутри элемента так, как мы делали это с двумерными трансформациями** (то есть посредством противоположной трансформации). Отменить трехмерную трансформацию на внутреннем элементе технически возможно, но чрезвычайно сложно. Следовательно, единственный практический способ воспользоваться преимуществом трехмерных трансформаций для создания трапеции — применить трансформацию к псевдоэлементу, по аналогии с подходом, который мы применили для параллелограммов в **секрете «Параллелограммы»**:

TRAPEZOID

TRAPEZOID

Рис. 3.42. Создание трапеции посредством трехмерного вращения

Наверху: до

Внизу: после

```
.tab {
    position: relative;
    display: inline-block;
    padding: .5em 1em .35em;
    color: white;
}

.tab::before {
    content: ''; /* Чтобы сгенерировать поле */
    position: absolute;
    top: 0; right: 0; bottom: 0; left: 0;
    z-index: -1;
    background: #58a;
    transform: perspective(.5em) rotateX(5deg);
}
```

TRAPEZOID

Рис. 3.43. Применение трехмерной трансформации к полю, сгенерированному с помощью псевдоэлемента, для того чтобы трансформация не распространялась на текст

Как видно на рис. 3.43, это позволяет создать простейшую трапециевидную фигуру. Но остается нерешенной еще одна проблема. Когда мы применяем трансформацию, не устанавливая значение `transform-origin`, элемент поворачивается в пространстве вокруг своего центра. Следовательно, его

габариты на двумерной проекции, которую мы видим на экране, меняются сразу в нескольких аспектах, что хорошо видно на рис. 3.44: он становится шире и немного смещается вверх, его высота чуть-чуть сокращается и т. п. Все это создает сложности в проработке дизайна.

Рис. 3.44. Наша трапеция, наложенная на изначальную, нетрансформированную версию элемента. Это позволяет увидеть, каким изменениям подвергается элемент

Для того чтобы получить больший контроль над размерами элемента, можно определить свойство `transform-origin: bottom;` — оно **фиксирует основание элемента при повороте его в пространстве**. Отличие от предыдущего варианта вращения вы видите на рис. 3.45. Теперь ситуация стала намного более предсказуемой: уменьшается только высота элемента. Однако это уменьшение намного более ярко выражено, так как весь элемент поворачивается в сторону от наблюдателя, тогда как в предыдущем варианте одна половина поворачивалась «за» экраном, а вторая — перед ним, и в трехмерном пространстве элемент в целом оказывался ближе к наблюдателю. Для того чтобы справиться с искажением, можно было бы увеличить забивку сверху. Однако в браузерах, не поддерживающих трехмерную трансформацию, результат будет выглядеть ужасающе (рис. 3.46). Вместо этого давайте **увеличим размер элемента посредством еще одной трансформации**, для того чтобы в случаях, когда трехмерные трансформации не поддерживаются, отменялись вообще все изменения формы объекта. Немного поэкспериментировав, можно убедиться, что небольшого масштабирования по вертикали (то есть трансформации `scaleY()`) — около 130% — достаточно, чтобы компенсировать потерянное пространство.

Рис. 3.45. Наша трапеция, наложенная на изначальную, нетрансформированную версию элемента. Это позволяет увидеть, каким изменениям подвергается элемент, когда мы используем `transform-origin: bottom;`

Рис. 3.46. Попытка справиться с искажением путем увеличения высоты забивки приводит к тому, что резервное решение выглядит странно (наверху)

Конечный результат и резервное решение показаны на рис. 3.47. Сейчас результат визуально эквивалентен тому, который дает нам описанная выше старая добрая техника, основанная на рамках, — однако новый синтаксис намного более емкий. Превосходство данной техники становится очевидным, когда вы начинаете применять стилизацию ко вкладкам. Например, взгляните на следующий код, в котором мы определяем стили для вкладок с рис. 3.48:

```
nav > a {
    position: relative;
    display: inline-block;
    padding: .3em 1em 0;
}
nav > a::before {
    content: '';
    position: absolute;
    top: 0; right: 0; bottom: 0; left: 0;
    z-index: -1;
    background: #ccc;
    background-image: linear-gradient(
                        hsla(0,0%,100%,.6),
                        hsla(0,0%,100%,0));
    border: 1px solid rgba(0,0,0,.4);
    border-bottom: none;
    border-radius: .5em .5em 0 0;
    box-shadow: 0 .15em white inset;
    transform: perspective(.5em) rotateX(5deg);
    transform-origin: bottom;
}
```

Рис. 3.47. Восполняя утерянную высоту с помощью `scale()`, мы обеспечиваем намного лучшее резервное решение (наверху)



Рис. 3.48. Преимущество данной техники кроется в ее гибкости относительно стилей

Как вы видите, мы определили фоны, рамки, скругленные углы и тени для полей — и все это сразу заработало безо всяких плясок с бубнами! Кроме того, всего лишь изменив значение `transform-origin` на `bottom left` или `bottom right`, мы можем получить вкладки, скошенные соответственно влево или вправо! (Пример показан на рис. 3.49.)

Несмотря на все ее добродетели, идеальной данную технику все же не назовешь. У нее есть один крупный недостаток: угол наклона сторон зависит от ширины элемента. Поэтому при работе с содержимым переменной длины получить трапеции с одинаковыми углами становится очень сложно. И все же описанная техника прекрасно подходит для элементов, включающих лишь небольшие вариации ширины, таких как навигационное меню. При определении таких элементов различия почти не заметны.

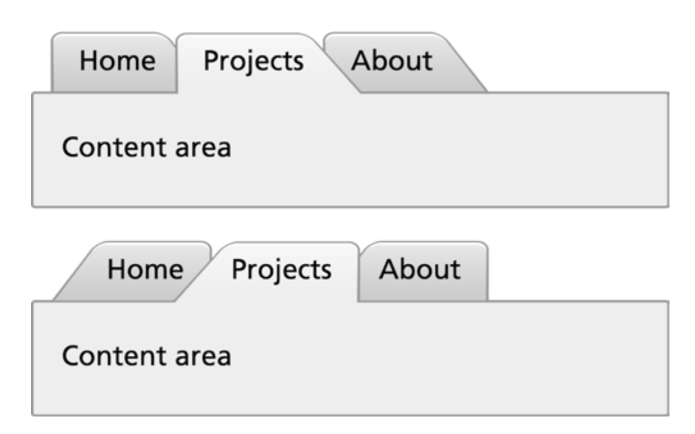


Рис. 3.49. Скошенные вкладки получились благодаря другим значениям `transform-origin`

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**trapezoid-tabs**

СВЯЗАННЫЕ СПЕЦИФИКАЦИИ

CSS Transforms: http://w3.org/TR/css-transforms

14 Простые секторные диаграммы

ПРЕДВАРИТЕЛЬНЫЕ ТРЕБОВАНИЯ

Градиенты CSS, базовое знание формата SVG, анимация в CSS, **секрет «Фоны в полоску», секрет «Гибкие эллипсы»**

Проблема

Секторные диаграммы — даже в самой простой своей двухцветной форме — всегда славились сложностью создания с помощью веб-технологий, несмотря на широкое распространение и массу вариантов использования: от представления простых статистических данных до индикаторов прогресса и таймеров.

Реализации чаще всего означают создание нескольких изображений для разных значений секторной диаграммы во внешнем графическом редакторе или необходимость прибегать к помощи объемных каркасов JavaScript, предназначенных для куда более сложных диаграмм.

Хотя это уже и вышло из категории невозможного, простого однострочного решения для данной задачи пока что нет. Однако уже сегодня существует несколько хороших способов достичь нужного результата, обеспечивающих к тому же пригодный для дальнейшей поддержки CSS-код.

Решение на основе трансформации

Это решение оптимально в терминах разметки: оно требует создания всего лишь одного элемента, а остальное реализуется с помощью псевдоэлементов, трансформаций и градиентов CSS. Начнем с простого элемента:

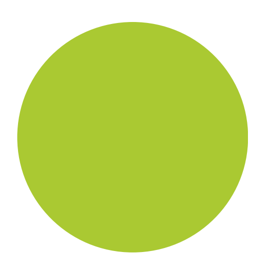


Рис. 3.50. Наша отправная точка (или секторная диаграмма, показывающая 0%)

HTML

```
<div class="pie"></div>
```

Пока предположим, что нам требуется секторная диаграмма, отображающая жестко закодированное значение **20%**. Мы поработаем над тем, чтобы сделать ее гибкой, чуть позже, а для начала применим стили, превращающие элемент в круг, который будет служить нашим фоном (рис. 3.50):

```
.pie {
    width: 100px; height: 100px;
    border-radius: 50%;
    background: yellowgreen;
}
```



Рис. 3.51. Окрашивание правой части нашего круга в коричневый цвет с помощью простого линейного градиента

Наша секторная диаграмма будет зеленой (точнее, цвета `yellowgreen`), а процентное значение на ней будет отображаться цветом `#655`. Первой идеей может быть использование трансформации скашивания для формирования сектора, соответствующего процентному значению, но несколько экспериментов быстро докажут, что решение получается слишком беспорядочным. Вместо этого мы закрасим левую и правую части нашего круга **двумя обозначенными выше цветами,** а затем с помощью **вращающегося псевдоэлемента откроем только небольшую часть, соответствующую требуемому процентному значению**.

Для того чтобы закрасить правую часть круга коричневым цветом, воспользуемся простым линейным градиентом:

```
background-image:
    linear-gradient(to right, transparent 50%, #655 0);
```

Как демонстрирует рис. 3.51, это все, что нам требуется. Теперь перейдем к стилизации псевдоэлемента, который будет служить маской на нашем круге:

```
.pie::before {
    content: '';
    display: block;
    margin-left: 50%;
    height: 100%;
}
```

На рис. 3.52 видно, где в данный момент находится наш псевдоэлемент относительно элемента, представляющего саму секторную диаграмму. Пока что с ним не связаны никакие стили, и он ничего не закрывает. Это просто невидимый прямоугольник. Прежде чем приступать к определению стилей, зафиксируем несколько наблюдений:

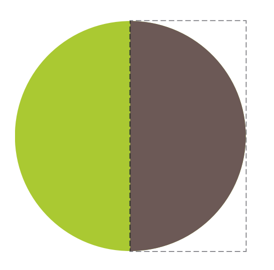


Рис. 3.52. Псевдоэлемент, который будет выполнять функцию маски, обозначен здесь пунктирными линиями 

- так как мы хотим, чтобы псевдоэлемент **закрывал коричневую часть круга**, мы должны определить для него зеленый фон с помощью `background-color: inherit`. Это поможет избежать дублирования, ведь нам требуется тот же фоновый цвет, что и у родительского элемента;
- мы будем **вращать псевдоэлемент вокруг центра круга**, который совпадает с серединой левой стороны псевдоэлемента, поэтому нам необходимо установить для него значение свойства `transform-origin`, равное `0 50%`, или же просто `left`;
- мы не хотим, чтобы псевдоэлемент имел форму прямоугольника, так как прямоугольник вылезает за границу секторной диаграммы. Следовательно, нам необходимо либо установить для `.pie` значение `overflow: hidden`, либо использовать подходящее значение `border-radius`, чтобы превратить его в полукруг.

Будьте осторожны — не напишите по ошибке `background: inherit;` вместо `background-color: inherit;`, иначе градиент также будет унаследован!

Собирая все вместе, получаем следующий CSS-код для нашего псевдоэлемента:

```
.pie::before {
    content: '';
    display: block;
    margin-left: 50%;
    height: 100%;
    border-radius: 0 100% 100% 0 / 50%;
    background-color: inherit;
    transform-origin: left;
}
```

Сейчас наша секторная диаграмма выглядит как на рис. 3.54. И здесь начинается главное веселье! Теперь мы можем вращать наш псевдоэлемент, применяя к нему трансформацию `rotate()`. Для `20%`, которые мы поставили себе целью нарисовать с самого начала, можно использовать значение `72deg` ($0,2 \times 360 = 72$) или просто `.2turn`, что намного понятнее и читабельнее. Как это работает с некоторыми другими значениями, вы можете увидеть на рис. 3.53.

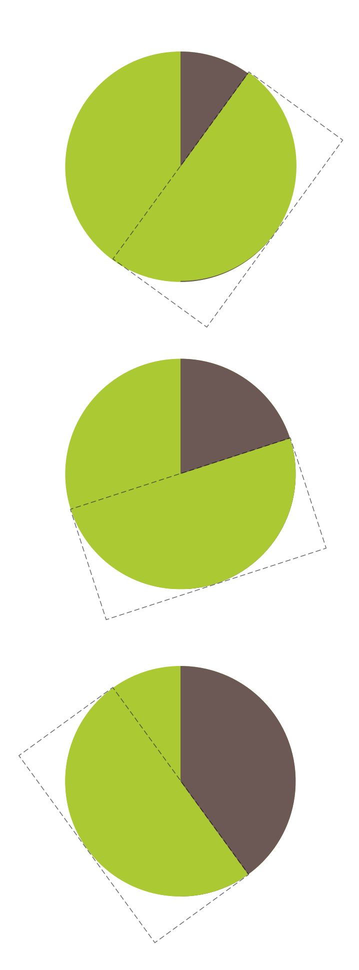


Рис. 3.53. Наша простая секторная диаграмма, отображающая разные процентные значения. Сверху вниз: 10% (`36deg` или `.1turn`), 20% (`72deg` или `.2turn`), 40% (`144deg` или `.4turn`)

Возможно, вы подумали, что задача решена, но не все так просто. Наша секторная диаграмма прекрасно отображает процентные значения от 0 до 50%, но когда мы пытаемся показать на ней значение 60% (применив вращение `.6turn`), происходит то, что вы видите на рис. 3.55. Однако не теряйте надежды! Это можно исправить, что мы сейчас и сделаем.

Если взглянуть на значения от `50%` до `100%` как на отдельную проблему, то легко заметить, что для нее можно использовать **инвертированную версию предыдущего решения**: коричневый псевдоэлемент, поворачивающийся от `0` до `.5turn` соответственно. Таким образом, для сектора размером 60% код псевдоэлемента выглядит так:

```
.pie::before {
    content: '';
    display: block;
    margin-left: 50%;
    height: 100%;
    border-radius: 0 100% 100% 0 / 50%;
    background: #655;
    transform-origin: left;
    transform: rotate(.1turn);
}
```

Результат этого действия можно видеть на рис. 3.56. И так как мы теперь умеем изображать любые процентные значения, мы можем даже **анимировать секторную диаграмму между 0% и 100%** с использованием средств анимации CSS, создав, таким образом, **привлекательный индикатор прогресса**:

```
@keyframes spin {
    to { transform: rotate(.5turn); }
}

@keyframes bg {
    50% { background: #655; }
}

.pie::before {
    content: '';
```

```
    display: block;
    margin-left: 50%;
    height: 100%;
    border-radius: 0 100% 100% 0 / 50%;
    background-color: inherit;
    transform-origin: left;
    animation: spin 3s linear infinite,
               bg 6s step-end infinite;
}
```

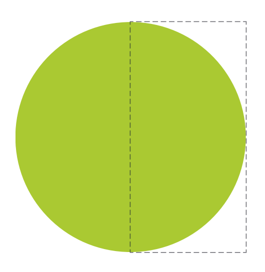


Рис. 3.54. Наш псевдоэлемент (обозначенный здесь пунктирным контуром) после того, как мы закончили определение его стилей

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**pie-animated**

Все это хорошо, но как определить стили для нескольких статичных секторных диаграмм с разными процентными значениями, то есть для самого распространенного сценария их использования? В идеальном случае нам хотелось бы иметь возможность напечатать простейший код вроде этого:

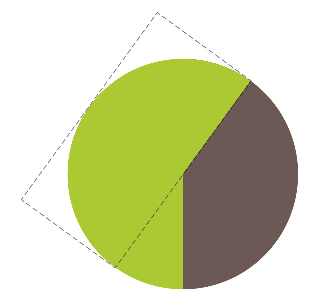


Рис. 3.55. Наша секторная диаграмма ломается на значениях, превышающих `50%` (показанное здесь соответствует значению `60%`)

HTML

```
<div class="pie">20%</div>
<div class="pie">60%</div>
```

...и получить две секторные диаграммы, одна из которых показывает `20%`, а вторая — `60%`. Сначала мы посмотрим, какие нам нужны **строковые стили**, а затем напишем короткий сценарий для разбора текстового содержимого и добавления этих строковых стилей. Это обеспечит **элегантный код**, **инкапсуляцию**, хорошую **сопровождаемость** и, что наиболее важно, **доступность**.

Проблема управления процентными значениями секторных диаграмм с помощью строковых стилей заключается в том, что CSS-код, отвечающий за установку процента, связан с псевдоэлементом. Как вы уже знаете, **невозможно определять строковые стили на псевдоэлементах,** поэтому нам **понадобится проявить изобретательность**.

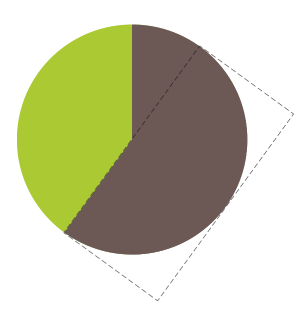


Рис. 3.56. Наша исправленная диаграмма и значение 60%

Решение обнаруживается в одном из самых неожиданных мест. Мы будем использовать анимацию,

которую уже представили выше, но **поставив ее на паузу**. Вместо того чтобы прокручивать ее как нормальную анимацию, мы будем использовать **отрицательные значения задержки**, чтобы иметь возможность **статически прогонять ее до любой желаемой точки** и останавливать в получившемся состоянии. Звучит непонятно? Да, отрицательные значения `animation-delay` не только допускаются спецификацией, но и чрезвычайно полезны в ситуациях, подобных этой.

> Отрицательная задержка **допустима**. Аналогично задержке в `0s`, это означает, что анимация запускается немедленно, но автоматически прокручивается вперед на абсолютное значение задержки, как если бы воспроизведение началось указанный период времени назад. Таким образом создается впечатление, что анимация воспроизводится не с начала, а с определенной точки после прохождения части пути.
>
> **— CSS Animations Level 1**
> (http://w3.org/TR/css-animations/#animation-delay)

Так как наша анимация поставлена на паузу, **единственным кадром, который отобразится на экране**, будет первый кадр, определяемый нашим отрицательным значением `animation-delay`. Процентное значение на секторной диаграмме будет соответствовать доле, которую **значение** `animation-delay` **составляет от общей продолжительности анимации**. Например, с текущей длительностью анимации, равной `6s`, для отображения процентного значения `20%` нам потребуется установить для `animation-delay` значение `-1.2s`. Чтобы упростить вычисления, зададим длительность анимации, равную `100s`. Помните только, что **поскольку анимация ставится на паузу окончательно и никогда не возобновляется, определяемая нами общая длительность анимации ни на что больше не влияет**.

Осталась одна только последняя проблема: **анимация привязана к псевдоэлементу, а мы хотим задать строковый стиль для элемента** `.pie`. Но так как для `<div>` не определяется никакой анимации, мы можем установить `animation-delay` как строковый стиль для этого элемента, а затем использовать `animation-delay: inherit;` на псевдоэлементе. Складывая все вместе, получаем такую разметку для секторных диаграмм, соответствующих значениям `20%` и `60%`:

СОВЕТ

Вы можете применять эту технику и в других случаях, когда возникает необходимость использовать значения из диапазона без повторений и сложных вычислений, а также для отладки анимации путем пошагового прогона. Более простой, изолированный пример данной техники вы найдете на веб-странице http://play.csssecrets.io/static-interpolation.

HTML

```
<div class="pie"
    style="animation-delay: -20s"></div>
<div class="pie"
    style="animation-delay: -60s"></div>
```

А CSS-код для этой анимации из представленного выше становится таким (за исключением правила `.pie`, которое не меняется):

```
@keyframes spin {
    to { transform: rotate(.5turn); }
}

@keyframes bg {
    50% { background: #655; }
}

.pie::before {
    /* [Остальные стили не меняются] */
    animation: spin 50s linear infinite,
               bg 100s step-end infinite;
    animation-play-state: paused;
    animation-delay: inherit;
}
```

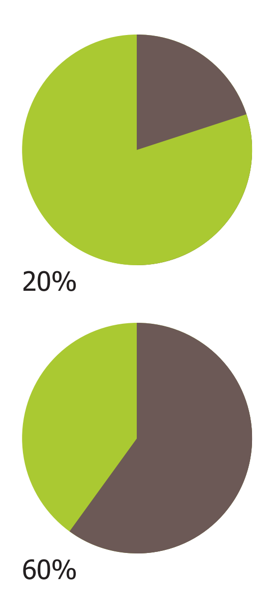


Рис. 3.57. Наш текст до того, как мы его спрячем

На этом этапе мы уже можем преобразовать разметку так, чтобы процентные значения использовались в качестве содержимого, как первоначально и планировалось, а также добавить строковые стили `animation-delay` с помощью простого сценария:

JS

```
$$('.pie').forEach(function(pie) {
    var p = parseFloat(pie.textContent);
    pie.style.animationDelay = '-' + p + 's';
});
```

Обратите внимание, что текст мы менять не стали, так как он нужен нам для обеспечения **доступности** и **удобства использования**. В данный момент наши секторные диаграммы выглядят как на рис. 3.57. Нам нужно скрыть текст, что можно сделать, не теряя в доступности, посредством установки `color: transparent`; в этом случае текст все так же можно будет **выделить и напечатать**. В качестве последнего штриха **процентные значения можно выровнять по центру секторных диаграмм**, чтобы, когда пользователь выделяет содержимое, они не оказывались в произвольных местах страницы. Для этого необходимо следующее:

- преобразовать свойство `height` диаграммы в `line-height` (или добавить значение `line-height`, равное `height`, но это бессмысленное дублирование кода, поскольку `line-height` все равно установит точно такое же вычисленное значение);
- задать размер и позицию псевдоэлемента посредством **абсолютного позиционирования**, чтобы текст не выталкивался вниз;
- добавить `text-align: center;` для центрирования текста по горизонтали.

Финальная версия кода выглядит так:

```
.pie {
    position: relative;
    width: 100px;
    line-height: 100px;
    border-radius: 50%;
    background: yellowgreen;
    background-image:
        linear-gradient(to right, transparent 50%, #655 0);
    color: transparent;
    text-align: center;
}

@keyframes spin {
    to { transform: rotate(.5turn); }
}

@keyframes bg {
    50% { background: #655; }
}

.pie::before {
    content: '';
    position: absolute;
    top: 0; left: 50%;
    width: 50%; height: 100%;
    border-radius: 0 100% 100% 0 / 50%;
    background-color: inherit;
    transform-origin: left;
    animation: spin 50s linear infinite,
               bg 100s step-end infinite;
    animation-play-state: paused;
    animation-delay: inherit;
}
```

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**pie-static**

Решение на основе SVG

Формат SVG упрощает решение множества задач, связанных с графиками, и секторные диаграммы не исключение. Однако вместо того чтобы создавать секторные диаграммы с помощью контуров, что потребовало бы сложных вычислений, мы воспользуемся небольшим трюком.

Начнем с круга:

SVG

```
<svg width="100" height="100">
<circle r="30" cx="50" cy="50" />
</svg>
```

Как вы, вероятно, знаете, эти свойства CSS также доступны как атрибуты элемента SVG, что может быть более практичным решением, если одна из ваших задач — обеспечение переносимости кода.

Теперь применим к нему простейшую стилизацию:

```
circle {
    fill: yellowgreen;
    stroke: #655;
    stroke-width: 30;
}
<P_158_01.eps>
```

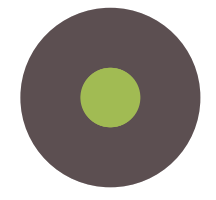


Рис. 3.58. Наша отправная точка: определенный с помощью SVG зеленый круг с жирной обводкой цвета `#655`

Наш круг с обводкой показан на рис. 3.58. Обводка в SVG включает не только свойства `stroke` и `stroke-width`. Существует множество других, менее известных свойств, позволяющих тонко настраивать представление обводки. Одно из них — `stroke-dasharray`, предназначенное для создания пунктирных обводок. Например, мы могли бы использовать его так:

```
stroke-dasharray: 20 10;
```

Это означает, что нам нужны штрихи длиной `20` и размер промежутка между ними, равный `10`, как на рис. 3.59. Сейчас вы, наверное, задаетесь вопросом, что общего может быть у этой иллюстрации использования обводки в SVG с секторными диаграммами. Понятнее станет, когда мы создадим обводку с нулевой шириной и зазором, равным или превышающим длину окружности нашего круга ($C = 2\pi r$, так что в нашем случае $C = 2\pi \times 30 \approx 189$):

```
stroke-dasharray: 0 189;
```



Рис. 3.59. Простая пунктирная обводка, созданная с помощью `stroke-dasharray`

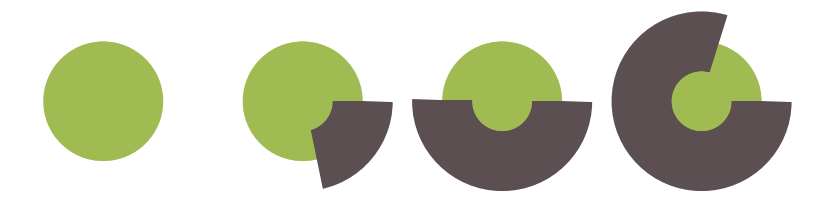


Рис. 3.60. Несколько значений `stroke-dasharray` и соответствующий визуальный результат. *Слева направо*: 0 189 40 189 95 189 150 189



Рис. 3.61. Наша SVG-графика начинает напоминать секторную диаграмму

Помните: обводка в SVG всегда находится наполовину внутри, а наполовину снаружи элемента, с которым она связана. В будущем мы получим возможность контролировать это поведение.

Как демонстрирует первый круг на рис. 3.60, такой код **полностью убирает любую обводку**, и нам остается только зеленый круг. Но самое интересное начинается, когда мы принимаемся **увеличивать первое значение** (см. рис. 3.60): поскольку промежуток так велик, в результате мы получаем не пунктирную обводку, а лишь один из штрихов обводки, который покрывает заданную часть окружности.

Вероятно, вы уже догадались, к чему я клоню: если мы уменьшим радиус нашего круга так, чтобы **обводка полностью покрывала его**, то получится фигура, очень сильно напоминающая секторную диаграмму. Например, на рис. 3.61 вы видите результат для случая, когда радиус круга равен 25, а значение `stroke-width` — 50, как определяется в следующем фрагменте кода:

SVG

```
<svg width="100" height="100">
    <circle r="25" cx="50" cy="50" />
</svg>

circle {
    fill: yellowgreen;
    stroke: #655;
    stroke-width: 50;
    stroke-dasharray: 60 158; /* 2π × 25 ≈ 158 */
}
```

Превратить это теперь в секторную диаграмму, подобную тем, которые мы создавали с помощью предыдущих решений, довольно просто: нам нужно всего лишь добавить **зеленый круг большего радиуса прямо под обводкой**

и **повернуть обводку на 90° против часовой стрелки**, чтобы она начиналась наверху посередине. Так как элемент `<svg>` — это также элемент HTML, мы можем просто определить для него следующие стили:

```
svg {
    transform: rotate(-90deg);
    background: yellowgreen;
    border-radius: 50%;
}
```

Финальный результат показан на рис. 3.62. С помощью этой техники создать анимацию секторной диаграммы от `0%` до `100%` еще проще. Нужно всего лишь анимировать средствами CSS свойство `stroke-dasharray` от `0 158` до `158 158`:

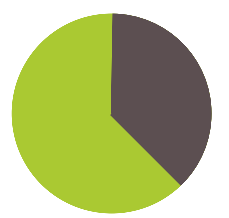


Рис. 3.62. Готовая секторная диаграмма, созданная с помощью SVG

```
@keyframes fillup {
    to { stroke-dasharray: 158 158; }
}

circle {
    fill: yellowgreen;
    stroke: #655;
    stroke-width: 50;
    stroke-dasharray: 0 158;
    animation: fillup 5s linear infinite;
}
```

В качестве дополнительного усовершенствования мы можем задать точный радиус круга, такой, чтобы **длина его окружности была равна 100** (вернее, составляла бесконечно близкое к 100 значение). Тогда мы сможем задавать значения `stroke-dasharray` в процентах, избегая необходимости выполнять вычисления. Поскольку длина окружности равна $2\pi r$, нам требуется радиус, равный $100/2\pi \approx 15{,}915494309$, что для наших целей можно округлить до 16. Также мы зададим габаритные размеры для элемента SVG с помощью атрибута `viewBox` вместо атрибутов `width` и `height`, чтобы его размер автоматически корректировался в зависимости от размеров контейнера.

После всех этих модификаций разметка для секторной диаграммы с рис. 3.62 будет выглядеть так:

SVG

```
<svg viewBox="0 0 32 32">
    <circle r="16" cx="16" cy="16" />
</svg>
```

А CSS-код нам потребуется такой:

```
svg {
    width: 100px; height: 100px;
    transform: rotate(-90deg);
    background: yellowgreen;
    border-radius: 50%;
}

circle {
    fill: yellowgreen;
    stroke: #655;
    stroke-width: 32;
    stroke-dasharray: 38 100; /* для 38% */
}
```

Обратите внимание, как **легко теперь поменять процентное значение**. Но, конечно, даже после такого упрощения нам не хочется повторять всю эту SVG-разметку для каждой секторной диаграммы. Настало время обратиться за помощью к JavaScript и привнести в решение немного автоматизации. Мы напишем небольшой сценарий, который будет брать простую HTML-разметку, подобную следующей...

HTML

```
<div class="pie">20%</div>
<div class="pie">60%</div>
```

...и добавлять строковый SVG внутри каждого элемента `.pie` со всеми необходимыми элементами и атрибутами. Кроме того, он будет добавлять элемент `<title>` для обеспечения **доступности**, чтобы программы чтения экрана также могли понимать, какое процентное значение отображается на каждой диаграмме. Готовый сценарий будет таким:

JS

```
$$('.pie').forEach(function(pie) {
    var p = parseFloat(pie.textContent);
    var NS = "http://www.w3.org/2000/svg";
    var svg = document.createElementNS(NS, "svg");
    var circle = document.createElementNS(NS, "circle");
    var title = document.createElementNS(NS, "title");
    circle.setAttribute("r", 16);
    circle.setAttribute("cx", 16);
    circle.setAttribute("cy", 16);
    circle.setAttribute("stroke-dasharray", p + " 100");
    svg.setAttribute("viewBox", "0 0 32 32");
```

```
    title.textContent = pie.textContent;
    pie.textContent = '';
    svg.appendChild(title);
    svg.appendChild(circle);
    pie.appendChild(svg);
});
```

Вот и всё! Не исключено, что **вы считаете, что метод с CSS лучше**, потому что код проще и выглядит менее инопланетным. Но у **метода с SVG есть определенные преимущества**, которые решение на чистом CSS обеспечить не в состоянии:

- **третий цвет добавить очень просто**: всего лишь создайте еще один круг с обводкой и сместите его обводку с помощью `stroke-dashoffset`. Или же

БУДУЩЕЕ.
СЕКТОРНЫЕ ДИАГРАММЫ

Помните конические градиенты из **секрета «Шахматные доски»?** Здесь их помощь также была бы неоценимой. Все, что нам потребовалось бы для создания секторной диаграммы — это круглый элемент с коническим градиентом, включающим две границы перехода цвета. Например, секторную диаграмму для значения 40%, показанную на рис. 3.53, можно было бы определить с помощью такого простого кода:

```
.pie {
width: 100px; height: 100px;
border-radius: 50%;
background: conic-gradient(#655 40%, yellowgreen 0);
           }
```

Помимо этого, после того как будет повсеместно реализована обновленная функция `attr()`, определенная в **CSS Values Level 3** (http://w3.org/TR/css3-values/#attr-notation), мы сможем контролировать процентное значение с помощью простого атрибута HTML:

```
background: conic-gradient(#655 attr(data-value %),
            yellowgreen 0);
```

Это также невероятно упрощает задачу добавления третьего цвета. Например, для создания секторной диаграммы, аналогичной той, которая отображается наверху этого поля, мы бы могли всего лишь добавить еще две границы перехода цвета:

```
background: conic-gradient(deeppink 20%, #fb3 0, #fb3 30%,
            yellowgreen 0);
```

прибавьте его длину штриха к длине штриха предыдущего круга (находящегося под ним). А как вы представляете себе добавление третьего цвета на секторные диаграммы в первом решении?

- **не приходится задумываться о проблемах с печатью страницы**, так как элементы SVG считаются содержимым и печатаются точно так же, как элементы `<img>`. Первое решение зависит от определения фона, и, следовательно, распечатать такую диаграмму невозможно;
- мы можем **менять цвета с помощью строковых стилей**, что означает, что они поддаются легкой настройке посредством **сценариев** (например, могут учитывать **значения, введенные пользователем**). Первое решение полагается на псевдоэлементы, которые не поддерживают строковые стили (кроме как через наследование), что не всегда удобно.

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**pie-svg**

СВЯЗАННЫЕ СПЕЦИФИКАЦИИ

CSS Transforms: http://w3.org/TR/css-transforms

CSS Image Values: http://w3.org/TR/css-images

CSS Backgrounds & Borders: http://w3.org/TR/css-backgrounds

Scalable Vector Graphics: http://w3.org/TR/SVG

CSS Image Values Level 4: http://w3.org/TR/css4-images

4

Визуальные эффекты

15 Односторонние тени

Проблема

Один из наиболее часто задаваемых вопросов относительно свойства `box-shadow`, которые я получаю на веб-сайтах с вопросами и ответами, — как создать тень только с одной стороны (или, реже, только с двух). Быстрый поиск по http://stackoverflow.com возвращает около тысячи результатов по данному запросу. И ничего удивительного, ведь отображение тени только с одной стороны создает более утонченный, но настолько же реалистичный эффект. Часто потерявшие надежду разработчики даже отправляют сообщения в список рассылки рабочей группы CSS, запрашивая создание новых свойств вроде `box-shadow-bottom`, которые могли бы обеспечить такой результат. Однако подобные эффекты уже возможны при умном использовании старого доброго свойства `box-shadow`, которое мы все знаем и любим.

Тень с одной стороны

Большинство разработчиков используют `box-shadow` с тремя числовыми значениями и цветом, например, так:

```
box-shadow: 2px 3px 4px rgba(0,0,0,.5);
```

Следующая последовательность шагов хорошо (хотя и не совсем точно с технической точки зрения) описывает процесс создания такой тени (рис. 4.1):

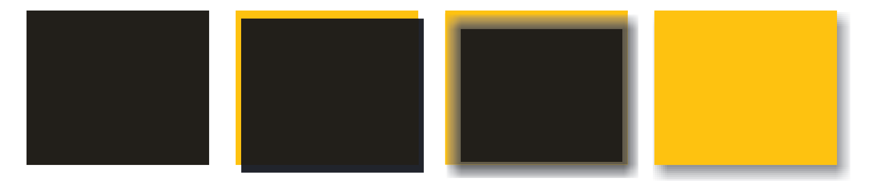


Рис. 4.1. Пример ментальной модели визуализации box-shadow

1. Рисуется прямоугольник цвета `rgba(0,0,0,.5)` с теми же габаритными размерами и в той же позиции, что и наш элемент.
2. Он смещается на `2px` вправо и на `3px` вниз.
3. Он размывается на `4px` с помощью алгоритма размытия Гаусса (или схожего). По сути, это означает, что цветовой переход по краям тени между цветом тени и полной прозрачностью будет приблизительно в два раза больше радиуса размытия (в нашем примере это `8px`).
4. Размытый прямоугольник затем **обрезается по контуру пересечения с нашим исходным элементом** для создания впечатления, что он находится позади него. Это немного отличается от того, как большинство разработчиков визуализируют тени в уме (размытый прямоугольник под элементом). Однако в некоторых сценариях использования важно понимать, что **никакая тень под элементом не рисуется**. Например, если мы зададим для элемента полупрозрачный фон, то тени внизу мы не увидим. В этом отличие такой тени от `text-shadow`, которая не обрезается по контуру текста.

Если не указано иное, говоря о габаритных размерах элемента, мы имеем в виду габаритные размеры его *поля рамки*, а **не** его ширину и высоту, указанные в CSS-коде.

Точнее, мы увидим тень шириной `1px` наверху (`4px - 3px`), шириной `2px` слева (`4px - 2px`), шириной `6px` справа (`4px + 2px`) и шириной `7px` внизу (`4px + 3px`). На практике она будет казаться меньше, так как цветовые переходы по краям нелинейные — аналогично градиентам.

Использование радиуса размытия, равного `4px`, означает, что габаритные размеры нашей тени приблизительно на `4px` больше габаритных размеров элемента, поэтому часть тени будет выглядывать со всех сторон элемента. Мы могли бы установить другие значения сдвига, увеличив их как минимум на `4px`, чтобы спрятать тень сверху и слева. Но в результате у нас получится слишком уж бросающаяся в глаза тень, а это выглядит непривлекательно (рис. 4.2). Кроме того, даже если бы это нас не беспокоило, изначально мы все же хотели получить тень только с одной стороны, помните?



Рис. 4.2. Попытка спрятать тень наверху и слева с помощью сдвигов, равных по величине радиусу размытия

Решение предлагает нам менее известный четвертый числовой параметр, который указывается после радиуса размытия и носит название *радиуса размазывания*. **Радиус размазывания увеличивает или (если он меньше нуля) уменьшает размер тени на**



Рис. 4.3. Тень box-shadow только у нижней стороны элемента

указанное вами значение. Например, радиус размазывания, равный -5px, уменьшает ширину и высоту тени на 10px (по 5px с каждой стороны).

Отсюда логически вытекает, что если мы применим отрицательный радиус размазывания, абсолютное значение которого совпадает с радиусом размытия, то тень получит габаритные размеры, точно совпадающие с габаритными размерами элемента, для которого она определена. И если мы не будем двигать ее с помощью атрибутов смещения (первые два значения), то **эту тень совершенно не будет видно**. Следовательно, положительное значение смещения по вертикали позволит нам увидеть тень у нижней кромки элемента, но не вдоль остальных сторон — как раз тот эффект, которого мы пытались достичь:

```
box-shadow: 0 5px 4px -4px black;
```

Результат вы можете видеть на рис. 4.3.

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**shadow-one-side**

Тень вдоль двух соседних сторон

Еще один часто задаваемый вопрос касается создания тени с двух сторон элемента. Если это соседние стороны (например, правая и нижняя), то все просто: можно либо удовлетвориться эффектом, аналогичным показанному на рис. 4.2, либо воспользоваться вариантом трюка из предыдущего раздела, но с некоторыми отличиями:

- сжимая тень, мы не должны пытаться убрать размытие с обеих сторон — только с одной. Следовательно, для радиуса размазывания необходимо указать значение, не противоположное радиусу размытия, а равное лишь его половине (с противоположным знаком);
- нам нужны оба смещения, так как тень необходимо сдвинуть и по горизонтали, и по вертикали. Значения смещения должны быть больше или равны радиусу размытия, поскольку вдоль остальных двух сторон тень должна быть полностью спрятана.

Например, вот что нужно для создания черной (цвета black) тени шириной 6px вдоль правой и нижней сторон элемента:

```
box-shadow: 3px 3px 6px -3px black;
```

Результат показан на рис. 4.4.



Рис. 4.4. Тень box-shadow только вдоль двух соседних сторон

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**shadow-2-sides**

Тень вдоль двух противоположных сторон

Ситуация становится еще запутаннее, когда у нас возникает необходимость создать тени с двух противоположных сторон, например слева и справа. Так как радиус размазывания применяется одинаково ко всем сторонам (то есть невозможно указать, что мы хотим увеличить тень по горизонтали, но сжать по вертикали), единственный способ решить задачу — использовать две тени, по одной с каждой стороны. В остальном это тот же трюк, что и рассмотренный выше в **секрете «Тень с одной стороны»**, только примененный дважды:

В рабочей группе CSS ведутся обсуждения относительно того, стоит ли в будущем разрешить указывать отдельные значения радиуса размазывания по горизонтали и по вертикали. Это упростило бы решение данной задачи.

```
box-shadow: 5px 0 5px -5px black,
           -5px 0 5px -5px black;
```

Результат вы можете видеть на рис. 4.5.

Рис. 4.5. Тень box-shadow вдоль двух противоположных сторон

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**shadow-opposite-sides**

СВЯЗАННЫЕ СПЕЦИФИКАЦИИ

CSS Backgrounds & Borders: http://w3.org/TR/css-backgrounds

16 Падающие тени неправильной формы

ПРЕДВАРИТЕЛЬНЫЕ ТРЕБОВАНИЯ

Знание свойства `box-shadow`

Проблема

Свойство `box-shadow` прекрасно работает, когда нам требуется тень, отбрасываемая прямоугольником или любой другой фигурой, которую можно создать с помощью `border-radius` (несколько примеров вы найдете в **секрете «Гибкие эллипсы»**). Однако от него гораздо меньше пользы, когда мы работаем с псевдоэлементами или другими полупрозрачными вариантами декорирования, потому что `box-shadow` бессовестно игнорирует прозрачность. Приведу несколько примеров:

- полупрозрачные изображения, фоновые изображения и рамки, созданные с применением `border-image` (например, винтажная рама с позолотой);
- штрихпунктирные рамки, рамки с точечным пунктиром и полупрозрачные рамки без фона (или со значением `background-clip`, отличным от `border-box`);
- облачко с текстом, указатель для которого создан с помощью псевдоэлемента;
- скошенные углы, аналогичные тем, которые мы учились делать в **секрете «Срезанные углы»**;
- большинство эффектов загнутого уголка, включая описанный далее в этой главе;
- контуры, создаваемые с помощью `clip-path`, например ромбовидные изображения из **секрета «Изображения в форме ромба»**.

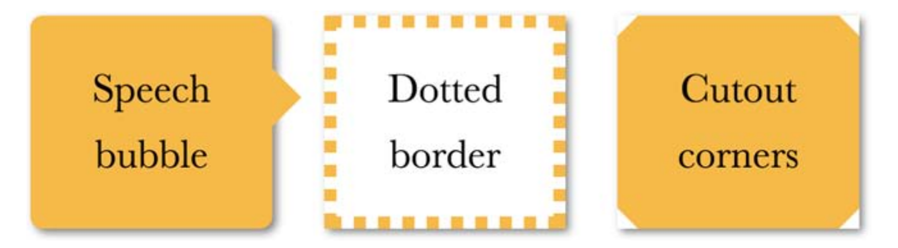
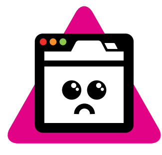


Рис. 4.6. Элементы, стилизованные с помощью CSS, с которыми использование box-shadow теряет всякий смысл (здесь применяется значение свойства box-shadow, равное 2px 2px 10px rgba(0,0,0,.5))

Результаты тщетных попыток применить `box-shadow` в некоторых из перечисленных ситуаций показаны на рис. 4.6. Существует ли решение для подобных случаев или нам придется вообще отказаться от использования теней?

Решение

Спецификация **Filter Effects** (http://w3.org/TR/filter-effects) предлагает решение данной проблемы в форме нового свойства `filter`, позаимствованного из формата SVG. Однако несмотря на то что фильтры CSS — это, по сути, те же самые **фильтры SVG**, для их использования **никакого знания SVG не требуется**. Они определяются посредством нескольких удобных функций, таких как `blur()`, `grayscale()` и — барабанная дробь — `drop-shadow()`! Можно даже составить последовательность из нескольких фильтров, если того требует ситуация, разделив их пробелами, например так:



Ограниченная поддержка

```
filter: blur() grayscale() drop-shadow();
```

Фильтр `drop-shadow()` принимает те же параметры, что и базовое свойство `box-shadow`, то есть без радиуса размазывания, без ключевого слова `inset`, без разделенных запятыми определений нескольких теней. Например, вместо:

```
box-shadow: 2px 2px 10px rgba(0,0,0,.5);
```

мы бы написали:

```
filter: drop-shadow(2px 2px 10px rgba(0,0,0,.5));
```

Результат применения этого фильтра `drop-shadow()` к элементам с рис. 4.6 демонстрируется на рис. 4.7.



Рис. 4.7. Результат использования фильтра drop-shadow() с элементами с рис. 4.6



Поскольку алгоритмы размытия могут быть разными, вам может потребоваться отрегулировать значение размытия в зависимости от потребностей в конкретной задаче.

Лучше всего в фильтрах CSS то, что они обеспечивают **изящные резервные решения**: когда фильтры не поддерживаются, ничего не ломается, просто никакой эффект не применяется. Вы можете добиться **чуть лучшей поддержки браузерами**, используя **заодно фильтр SVG**, если перед вами стоит задача любыми способами заставить этот эффект работать в как можно большем количестве браузеров. Соответствующие фильтры SVG для каждой функции фильтрации вы найдете в спецификации **Filter Effects** (http://w3.org/TR/filter-effects). Можно одновременно использовать и фильтр SVG, и упрощенный аналог на CSS, позволяя каскадным стилям самим определять победителя:

```
filter: url(drop-shadow.svg#drop-shadow);
filter: drop-shadow(2px 2px 10px rgba(0,0,0,.5));
```

К сожалению, если SVG-фильтр содержится в отдельном файле, то он не поддается такой же простой настройке, как приятная, удобная в использовании функция прямо в CSS-коде, а строковая функция, в свою очередь, захламляет код. В файле параметры фиксированы, и иметь несколько файлов на случай, если нам понадобятся слегка различающиеся тени, непрактично. Мы могли бы использовать URI данных (что заодно сэкономило бы нам несколько запросов HTTP), но они также приводят к увеличению размера файла. Поскольку фильтр SVG используется для обеспечения обходного решения, имеет смысл использовать один или два варианта, даже для немного отличающихся фильтров `drop-shadow()`.

Кроме того, не следует забывать, что **отбрасывать тень будет любая непрозрачная область**, независимо от того, каким элементом она представлена, — даже текст (на прозрачном фоне), как вы уже видели на рис. 4.7. Возможно, вы думаете, что отменить этот эффект можно с помощью `text-shadow: none;`, но `text-shadow` — это отдельное свойство, не способное компенсировать влияние фильтра `drop-shadow()` на текст. А если вы используете `text-shadow` для создания

настоящей тени текста, то благодаря фильтру `drop-shadow()` у этой тени также появится своя тень, то есть **вы создадите тень тени**! Взгляните на следующий пример CSS-кода (и простите за безвкусный результат — я всего лишь пытаюсь продемонстрировать всю дикость этой проблемы):

```
color: deeppink;
border: 2px solid;
text-shadow: .1em .2em yellow;
filter: drop-shadow(.05em .05em .1em gray);
```



Рис. 4.8. Тени text-shadow также отбрасывают тень сквозь фильтр drop-shadow()

Пример визуализации этого кода вы видите на рис. 4.8: здесь показана как тень `text-shadow`, так и отбрасываемая ею тень `drop-shadow()`.

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**drop-shadow**

СВЯЗАННЫЕ СПЕЦИФИКАЦИИ

Filter Effects: http://w3.org/TR/filter-effects

17 Создание цветового тона

ПРЕДВАРИТЕЛЬНЫЕ ТРЕБОВАНИЯ

Цветовая модель HSL, свойство `background-size`

Проблема

Добавление цветового тона к изображению в оттенках серого (или к изображению, преобразованному в оттенки серого) — популярный и элегантный способ придания визуального единообразия группе фотографий, выполненных в совершенно непохожих стилях. Часто этот эффект применяется статически и убирается по событию `:hover` и/или при других вариантах взаимодействия с изображением.

Традиционно мы создаем две версии изображения в графическом редакторе и добавляем немного простого CSS-кода, задача которого — подменять одну версию другой. Этот подход работает, но он раздувает исходный код и требует дополнительных HTTP-запросов, а сопровождать такой веб-сайт — это настоящая головная боль. Представьте, что было принято решение изменить цвет, использующийся для создания этого эффекта. Вам придется перебрать все изображения и создать для каждого новую монохромную версию!

Другие подходы включают наложение полупрозрачного цвета поверх изображения или изменение степени непрозрачности изображения и наложение его на подложку сплошного цвета. Однако это не настоящий тон: в таком решении не только цвета изображения не преобразуются в оттенки целевого цвета, но и существенно снижается контрастность.

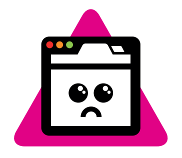


Рис. 4.9. На веб-сайте CSSConf 2014 этот эффект используется для оформления фотографий лекторов. Полноцветное изображение показывается при наведении указателя мыши и переводе фокуса на выбранный элемент

Также существуют сценарии, превращающие изображения в элемент `<canvas>` и применяющие тон средствами JavaScript. При этом действительно получается реальное тонированное изображение, но решение работает очень медленно и накладывает множество ограничений.

Согласитесь, было бы намного проще и удобнее применять цветовые тона к изображениям прямо в коде CSS?

Решение на основе фильтров



Ограниченная поддержка

Так как не существует единой функции фильтрации, разработанной специально для данного эффекта, нам придется проявить фантазию и скомбинировать **несколько фильтров**.

Первым мы применим фильтр `sepia()`, придающий изображению **ненасыщенный оранжево-желтый оттенок**, при котором тон большинства пикселов находится на уровне 35–40 (рис. 4.10). Если нам требовался именно этот цвет, то работа завершена. Но в большинстве случаев мы ставим себе целью





Рис. 4.10. *Наверху*: исходное изображение. *Внизу*: изображение после применения фильтра sepia()

добиться несколько иного результата. Если желаемый цвет насыщеннее, то увеличить насыщенность каждого пиксела можно с помощью фильтра `saturate()`. Предположим, что мы хотим придать изображению тон `hsl(335, 100%, 50%)`. Насыщенность нужно повысить совсем немного, поэтому мы используем параметр `4`. Точное значение зависит от каждого конкретного случая, поэтому ориентируйтесь на визуальный результат. Как демонстрирует рис. 4.11, этот комбинированный фильтр придает нашему изображению **теплый золотой тон**.

Как бы мило ни выглядело наше изображение, мы не планировали делать его ярким оранжево-желтым. Нам требуется глубокий ярко-розовый. Следовательно, необходимо также применить фильтр `hue-rotate()` для **смещения тона каждого пиксела на указанный угол в градусах**. Чтобы создать тон 335 из тона, приблизительно равного 40, необходимо добавить примерно 295 (335 – 40):



Рис. 4.11. Наше изображение после добавления фильтра saturate()

```
filter: sepia() saturate(4) hue-rotate(295deg);
```

Итак, мы придали тон нашему изображению, и готовый результат вы можете увидеть на рис. 4.12. Если этот эффект должен переключаться по наведению указателя мыши (`:hover`) или в зависимости от других состояний, то к нему можно также применить переходы CSS:

```
img {
    transition: .5s filter;
    filter: sepia() saturate(4) hue-
rotate(295deg);
}

img:hover,
img:focus {
    filter: none;
}
```

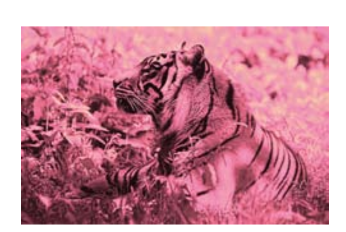
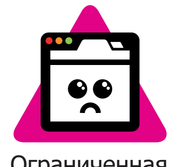


Рис. 4.12. Наше изображение после добавления третьего фильтра, hue-rotate()

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**color-tint-filter**

Решение на основе режимов смешивания



Решение с фильтрами работает, но вы, вероятно, заметили, что результат немного отличается от того, который мы бы получили в графическом редакторе. Хотя мы пытаемся для создания тона использовать очень яркий цвет, результат все равно выглядит немного **полинявшим**. Если мы попытаемся увеличить значение параметра фильтра `saturate()`, то получим **другой**, **чрезмерно стилизованный эффект**. К счастью, существует гораздо лучший подход: **режимы смешивания**!

Если вам когда-либо доводилось использовать графические редакторы, например Adobe Photoshop, то, вероятно, вы уже знакомы с режимами смешивания. Когда два элемента накладываются друг на друга, **режимы смешивания управляют тем**, **как цвета верхнего элемента смешиваются с цветами всего**, **что находится под ним**. Что касается придания цветового тона изображениям, для этого используется режим смешивания `luminosity`. Режим смешивания `luminosity` **сохраняет HSL-светлоту верхнего элемента**, **в то же время учитывая тон и насыщенность его подложки**. Если подложкой служит наш цвет, а элемент с этим режимом смешивания применяется к нашему изображению, то не означает ли это, что мы нашли ключ к созданию требуемого цветового тона?





Рис. 4.13. Сравнение метода на основе фильтров (*наверху*) и метода, основанного на режимах смешивания (*внизу*)

Для применения режима смешивания к элементу служат два свойства: `mix-blend-mode` применяет режимы смешивания **к элементам целиком**, а `background-blend-mode` применяет режимы смешивания **к каждому фоновому слою в отдельности**. Из этого следует, что использовать данный метод на изображении можно двумя способами, каждый из которых, к сожалению, не идеален:

- обернуть наше изображение в контейнер с желаемым фоновым цветом;
- использовать `<div>` вместо изображения, выбрав изображение, которому придается тон, в свойстве `background-image` и добавив внизу второй фоновый слой с желаемым цветом.

В зависимости от конкретного варианта использования можно прибегнуть к любому из этих подходов. Например, если нам нужно применить эффект к элементу `<img>`, то его необходимо обернуть другим элементом. Однако если у нас уже есть другой элемент, такой как `<a>`, то можно использовать его:

HTML

```
<a href="#something">
    <img src="tiger.jpg" alt="Rawrrr!" />
</a>
```

И тогда для применения эффекта понадобятся только два объявления:

```
a {
    background: hsl(335, 100%, 50%);
}

img {
    mix-blend-mode: luminosity;
}
```

Так же как и фильтры CSS, режимы смешивания обеспечивают изящные обходные пути: если они не поддерживаются, то никакие эффекты не применяются и изображение выводится в исходном виде.

Важно помнить, что в то время как **фильтры поддерживают анимацию**, **режимы смешивания анимировать невозможно**. Мы уже знаем, как создать анимационный эффект плавного перевода изображения в монохромный режим с помощью простого перехода CSS на свойстве `filter`. Реализовать это для режимов смешивания не получится. Но не беспокойтесь, это не означает, что вопрос анимации полностью отпадает. Просто нам придется применить нестандартное решение.

Как уже говорилось выше, `mix-blend-mode` смешивает элемент целиком со всем тем, что находится под ним. Следовательно, если мы применим режим смешивания `luminosity` посредством этого свойства, то изображение всегда будет смешиваться с **чем-то**. В то же время `background-blend-mode` смешивает каждый слой фонового изображения со слоями, находящимися ниже, и совершенно не в курсе происходящего за пределами элемента. Что произойдет, если мы создадим только одно фоновое изображение и прозрачный (`transparent`) фоновый цвет? Правильно: **никакого смешивания не будет**!

Воспользуемся этим наблюдением и применим свойство `background-blend-mode` с нашим эффектом. HTML-код немного изменится:

HTML

```
<div class="tinted-image"
    style="background-image:url(tiger.jpg)">
</div>
```

Теперь осталось лишь применить стилизацию CSS к этому единственному `<div>`, ведь данная техника не требует дополнительных элементов:

```
.tinted-image {
    width: 640px; height: 440px;
    background-size: cover;
    background-color: hsl(335, 100%, 50%);
    background-blend-mode: luminosity;
    transition: .5s background-color;
}

.tinted-image:hover {
    background-color: transparent;
}
```

Однако, как упоминалось выше, **ни одна из двух техник не идеальна**. Главные проблемы, возникающие при использовании данного подхода:

- габаритные размеры изображения необходимо жестко фиксировать в CSS-коде;
- **семантически** это не изображение, и программы чтения экрана не будут распознавать в нем изображение.

Как часто бывает в жизни, для этой задачи не существует безупречного решения, но в данном разделе мы изучили три разных пути создания желаемого эффекта, каждый со своими преимуществами и недостатками. Какой вы выберете — зависит от конкретных требований вашего проекта.

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**color-tint**

Спасибо Дадли Стори (Dudley Storey, http://demosthenes.info) за трюк с анимацией для режимов смешивания (http://demosthenes.info/blog/888/Create-Monochromatic-Color-Tinted-Images-With-CSS-blend).



Благодарности

СВЯЗАННЫЕ СПЕЦИФИКАЦИИ

Filter Effects: http://w3.org/TR/filter-effects

Compositing and Blending: http://w3.org/TR/compositing

CSS Transitions: http://w3.org/TR/css-transitions

18 Эффект матированного стекла

ПРЕДВАРИТЕЛЬНЫЕ ТРЕБОВАНИЯ

Цвета RGBA/HSLA

Проблема

Здесь мы используем термин «подложка» для обозначения той части страницы, которая находится под элементом и которая проглядывает сквозь его полупрозрачный фон.

Одним из первых вариантов использования полупрозрачных цветов было создание с их помощью фона поверх фотографий или других насыщенных деталями подложек, для того чтобы уменьшить контраст и улучшить читабельность текста. Результат получается довольно впечатляющим, но текст все равно бывает сложно прочитать, особенно если используются очень слабонасыщенные цвета и/или шумные подложки. Например, взгляните на рис. 4.14, на котором у главного элемента имеется полупрозрачный белый фон. Разметка выглядит так:

HTML

```
<main>
    <blockquote>
        "The only way to get rid of a temptation[…]"
        <footer>—
            <cite>
                Oscar Wilde,
                The Picture of Dorian Gray
            </cite>
        </footer>
    </blockquote>
</main>
```

А CSS-код выглядит так (для краткости все незначительные детали опущены):

```
body {
    background: url("tiger.jpg") 0 / cover fixed;
}

main {
    background: hsla(0,0%,100%,.3);
}
```



Рис. 4.14. На нашем полупрозрачном белом фоне текст прочитать трудно

Как вы видите, текст прочитать очень сложно, так как изображение яркое и содержит множество деталей, а фоновый цвет непрозрачен только на 25%. Разумеется, мы могли бы улучшить читабельность, увеличив параметр альфа-канала в определении фонового цвета, но тогда эффект будет не таким интересным (рис. 4.15).

В традиционном печатном дизайне эту проблему часто решают путем **размывания фрагмента фотографии**, **находящегося прямо под текстовым контейнером**. Размытые фоны не такие шумные, и, следовательно, текст поверх них читается намного проще. Поскольку эффект размытия дорогостоящ в терминах вычислительных ресурсов, в прошлом эту технику невозможно было использовать на веб-сайтах и в дизайне пользовательских интерфейсов. Однако графические процессоры непрерывно совершенствуются, а аппаратное ускорение становится доступным все в большем количестве разнообразных сценариев, поэтому сегодня мы сталкиваемся с эффектом размытия в оформлении интерфейсов довольно часто. За последние несколько лет нам довелось повстречать эту технику в новых версиях как Microsoft Windows, так и Apple iOS и Mac OS X (рис. 4.16).

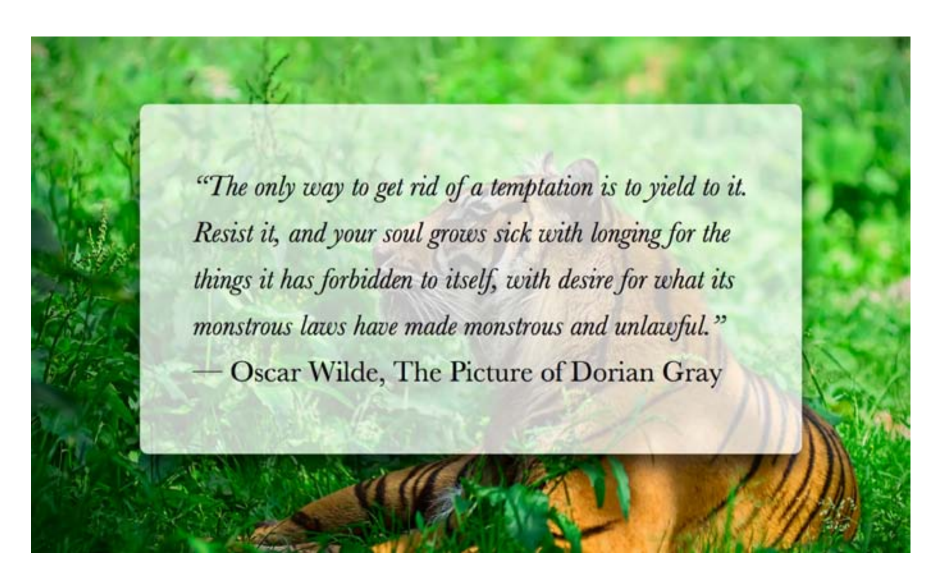


Рис. 4.15. Увеличение значения альфа-канала для фонового цвета решает проблему читабельности, но дизайн при этом теряет изюминку



Рис. 4.16. В последние годы полупрозрачные пользовательские интерфейсы с размытой подложкой встречаются все чаще, так как нагрузка на системные ресурсы, которую оказывает эффект размытия, перестала быть чрезмерно дорогой (слева — фрагмент интерфейса из Apple iOS 8.1; справа — фрагмент интерфейса из Apple OS X Yosemite)

В CSS у нас также есть возможность размывать элементы. Для этого используется фильтр `blur()`, по сути, представляющий собой аппаратно ускоренную версию соответствующего примитива фильтрации из SVG, который всегда был доступен в этом формате для размывания SVG-элементов. Однако если мы напрямую применим фильтр `blur()` в нашем примере, то размытию подвергнется весь элемент, что сделает его еще менее читабельным (рис. 4.17). Существует

ли способ применить эффект размытия только к подложке элемента (то есть к части фона, находящейся **позади** нашего элемента)?

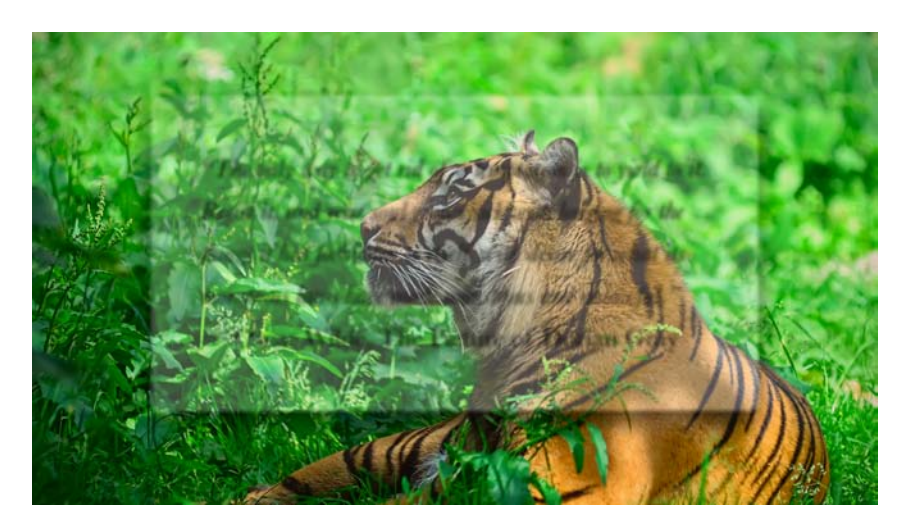


Рис. 4.17. Применение фильтра blur() к самому элементу только ухудшает ситуацию

Решение

При условии, что для нашего элемента определено свойство `background-attachment` со значением `fixed`, исправить ситуацию можно, проявив определенную изобретательность. Поскольку мы не можем применить размытие к самому элементу, **мы применим его к псевдоэлементу, расположенному позади элемента, фон которого полностью совпадает с фоном, определенным для** `<body>`.

Это возможно и с нефиксированными фонами, но с менее изящным решением.

Для начала добавим псевдоэлемент с абсолютным позиционированием и нулевыми смещениями, полностью покрывающий элемент `<main>`:

```
main {
    position: relative;
    /* [Остальные стили] */
}

main::before {
    content: '';
    position: absolute;
    top: 0; right: 0; bottom: 0; left: 0;
    background: rgba(255,0,0,.5); /* в целях отладки */
}
```

Будьте осторожны, используя отрицательные значения z-index для перемещения дочернего элемента под родительский: если этот родительский элемент вложен в другие элементы, для которых определены фоны, то они также окажутся поверх нашего дочернего элемента.

Мы также создали полупрозрачный красный фон (цвет red), для того чтобы видеть, что мы делаем, в противном случае отладка стилей для прозрачного (и, следовательно, невидимого) элемента становится слишком сложным делом. Как вы видите на рис. 4.18, в настоящий момент наш псевдоэлемент находится **поверх** содержимого, закрывая его. Это можно исправить, добавив z-index: -1; (рис. 4.20).



Рис. 4.18. Сейчас псевдоэлемент закрывает собой текст

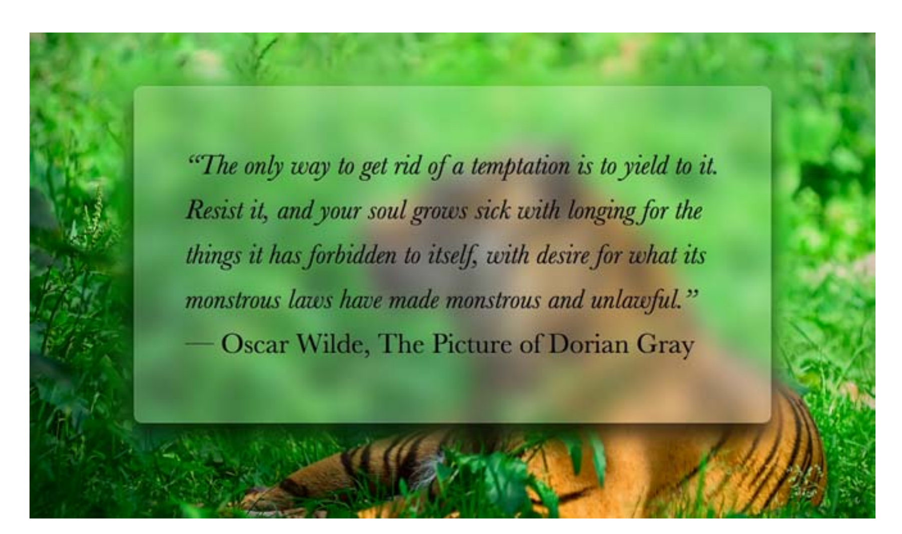


Рис. 4.19. Мы исправили проблему с затуханием размытия по краям, но теперь размывается часть изображения за пределами элемента

Теперь настало время заменить полупрозрачный красный фон другим фоном, в точности совпадающим с подложкой. Для этого можно либо скопировать фон из элемента `<body>`, либо выделить его в собственное правило. Можно ли теперь применять размытие? Давайте попробуем:

Почему бы просто не использовать `background: inherit` на `main::before`? Потому что в этом случае наследование будет выполнено от `main`, а не от `body`, и псевдоэлемент также получит полупрозрачный белый фон.

```
body, main::before {
    background: url("tiger.jpg") 0 / cover fixed;
}

main {
    position: relative;
    background: hsla(0,0%,100%,.3);
}

main::before {
    content: '';
    position: absolute;
    top: 0; right: 0; bottom: 0; left: 0;
    filter: blur(20px);
}
```

Как видно на рис. 4.21, мы почти достигли цели. Эффект размытия превосходно смотрится в центре, но чем ближе к краям, тем размытие слабее. Так происходит потому, что размывающий фильтр уменьшает область, покрытую сплошным

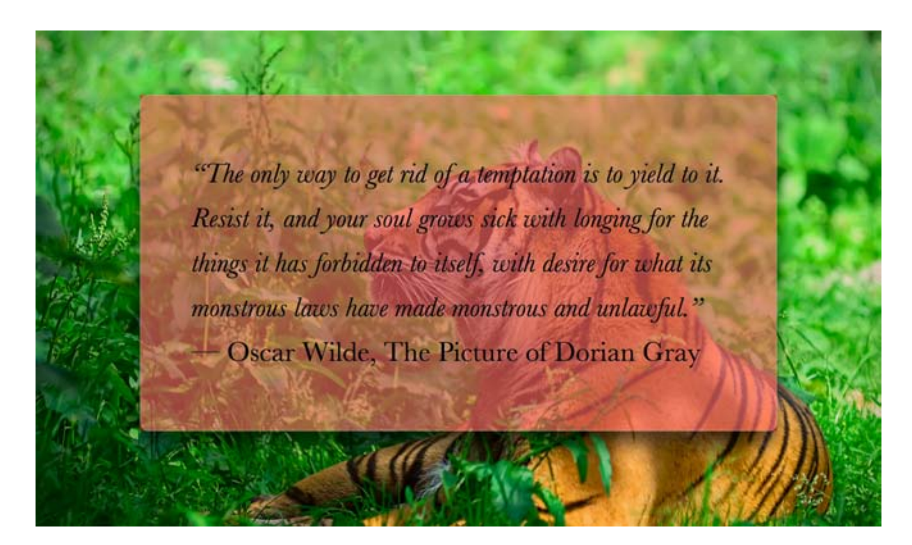


Рис. 4.20. С помощью z-index: -1; мы поставили псевдоэлемент позади его родительского элемента

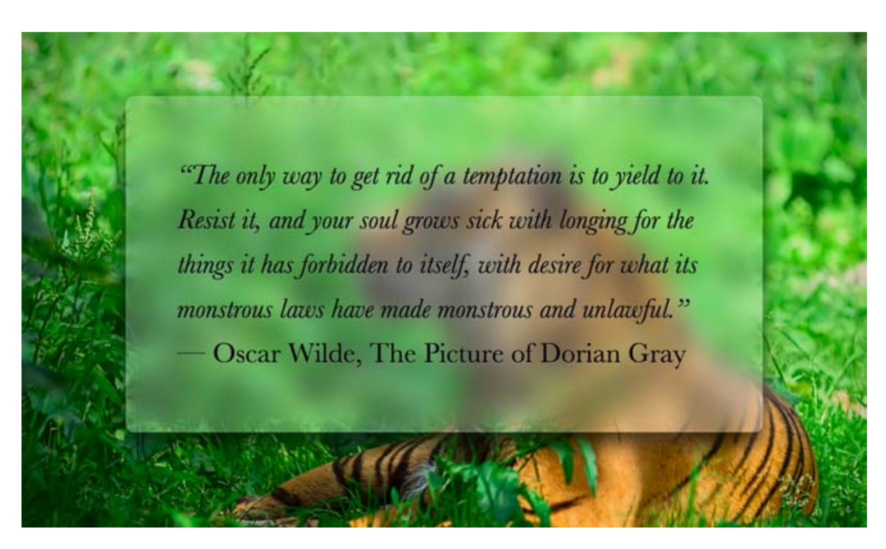


Рис. 4.21. Размытие псевдоэлемента работает почти идеально, но все же эффект ослабевает по краям, делая иллюзию матированного стекла менее достоверной

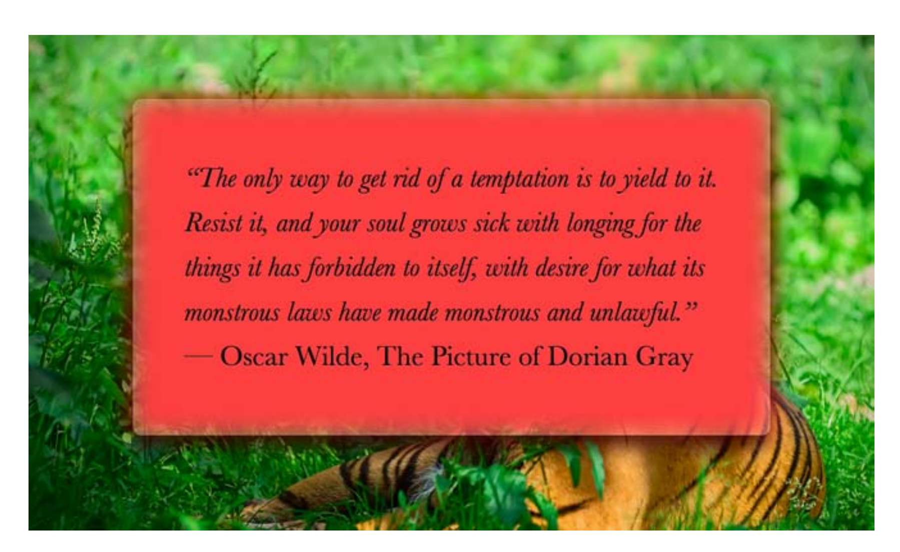


Рис. 4.22. Красный (red) фон помогает разобраться в происходящем

цветом, на значение, равное радиусу размытия. Снова применив красный (`red`) фон к нашему псевдоэлементу, мы сразу же поймем, в чем причина (рис. 4.22).

Чтобы обойти это ограничение, сделаем псевдоэлемент **по меньшей мере на** `20px` (радиус размытия) **больше габаритных размеров его контейнера**. Для этого установим размер полей, равный `-20px` или еще меньше, чтобы гарантированно

добиться нужного результата, так как в разных браузерах могут применяться разные алгоритмы размытия. Как демонстрирует рис. 4.19, это решает проблему с ослаблением размытия по краям, но теперь **размытие наблюдается за пределами нашего контейнера**, что делает его похожим на смазанное пятно вместо матированного стекла. К счастью, это легко поправить: мы всего лишь применим `overflow: hidden`; к элементу `main`, обрезав, таким образом, лишнее размытие. Финальная версия кода показана далее, а результат вы можете увидеть на рис. 4.23:

```
body, main::before {
    background: url("tiger.jpg") 0 / cover fixed;
}

main {
    position: relative;
    background: hsla(0,0%,100%,.3);
    overflow: hidden;
}

main::before {
    content: '';
    position: absolute;
    top: 0; right: 0; bottom: 0; left: 0;
    filter: blur(20px);
    margin: -30px;
}
```

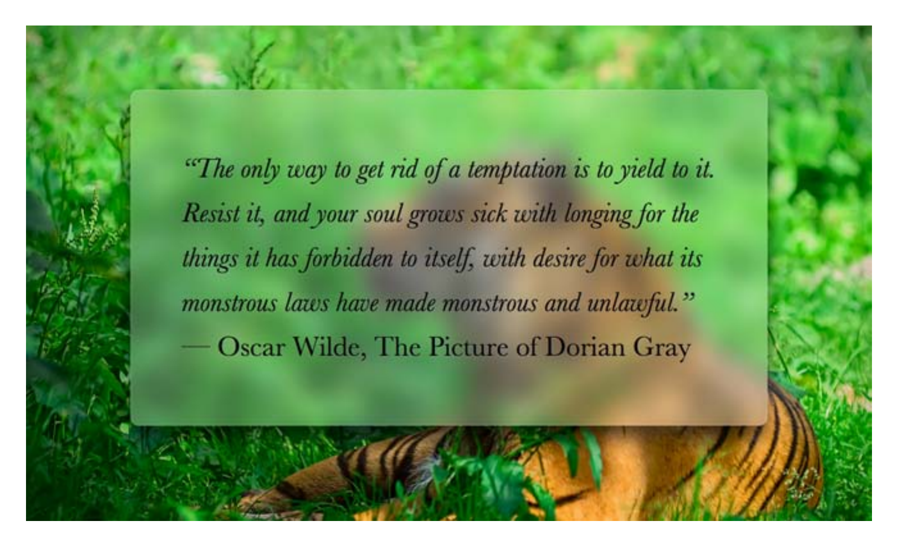


Рис. 4.23. Финальный результат

Обратите внимание, насколько более читабельной стала наша страница и как элегантно она теперь смотрится. Вопрос только в том, можно ли считать резервное решение для данного эффекта изящным выходом из ситуации. Если фильтры не поддерживаются, то мы получаем результат, который видели в начале (рис. 4.14). Для того чтобы сделать резервное решение более читабельным, можно увеличить степень непрозрачности фонового цвета.

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**frosted-glass**

СВЯЗАННЫЕ СПЕЦИФИКАЦИИ

Filter Effects: http://w3.org/TR/filter-effects

19 Эффект загнутого уголка

ПРЕДВАРИТЕЛЬНЫЕ ТРЕБОВАНИЯ

Трансформации CSS, градиенты CSS, секрет «Срезанные углы»

Проблема

Стилизация одного угла элемента (обычно верхнего правого или нижнего правого) так, чтобы он выглядел загнутым, с разной степенью реализма — это дизайнерский прием, не теряющий своей популярности вот уже много лет.

Сегодня в его применении нам помогают **несколько решений на чистом CSS**, первое из которых было опубликовано еще в 2010 году мастером работы с псевдоэлементами Николасом Галлахером (http://nicolasgallagher.com/pure-css-folded-corner-effect). Основной путь решения — добавить два треугольника в верхнем правом углу: один для представления загнутого уголка страницы и второй — закрывающий собой угол главного элемента. Эти треугольники чаще всего создаются с помощью проверенного временем трюка с рамкой.

В свое время эти решения смотрелись весьма впечатляюще, но сегодня мы понимаем, насколько они ограниченны. В некоторых ситуациях они попросту неприменимы:

- когда фон, находящийся позади нашего элемента, не залит сплошным цветом, а оформлен с использованием узора, текстуры, фотографии, градиента или фонового изображения любого другого типа;
- когда мы хотим загнуть уголок под другим углом, отличным от 45°, или же добавить легкое вращение.

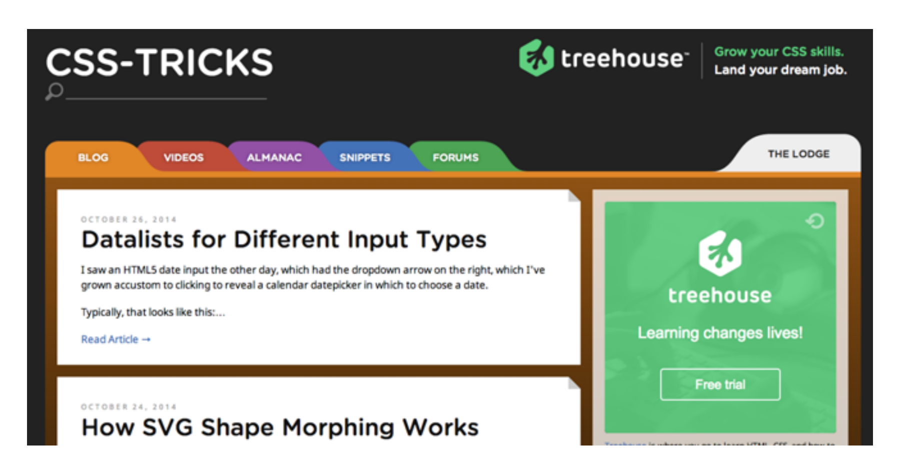


Рис. 4.24. В нескольких ранних версиях дизайна веб-сайта http://css-tricks.com загнутые уголки использовались для оформления верхнего правого угла полей, содержащих отдельные статьи сайта

Существует ли более гибкое решение для создания загнутых уголков с помощью CSS, которое не буксует в подобных случаях?

Решение для угла 45°

Для начала создадим элемент со скошенным верхним правым углом, воспользовавшись для этого решением на основе градиента из **секрета «Срезанные углы»**. Следующий код определяет скошенный угол размером `1em`, а графическое представление того, что должно получиться, вы видите на рис. 4.25:

```
background: #58a; /* Резервное решение */
background:
    linear-gradient(-135deg, transparent 2em, #58a 0);
```

«Единственный способ избавиться от искушения — это поддаться ему».
Оскар Уайлд,
«Портрет Дориана Грея»

Рис. 4.25. Наша отправная точка: элемент со срезанным верхним правым углом, для создания которого мы воспользовались градиентом

Дело наполовину сделано: все, что нам осталось, — это **добавить чуть более темный треугольник, представляющий загнутый уголок страницы**. Мы сделаем это с помощью еще одного градиента, а затем подгоним его размеры под наши требования с помощью `background-size` и поместим в **верхний правый угол**.

Для создания треугольника нам нужен расположенный под углом линейный градиент с двумя границами перехода цвета, встречающимися в середине:

```
background:
    linear-gradient(to left bottom,
        transparent 50%, rgba(0,0,0,.4) 0)
        no-repeat 100% 0 / 2em 2em;
```

Как выглядит результат, когда присутствует **только** этот фон, вы видите на рис. 4.26. На последнем шаге мы должны объединить эти два фрагмента кода, и на этом можно будет закончить, так? Давайте попробуем сделать это, убедившись, что треугольник, представляющий загнутый уголок, находится **поверх** градиента, создающего срезанный угол:

```
background: #58a; /* Резервное решение */
background:
    linear-gradient(to left bottom,
        transparent 50%, rgba(0,0,0,.4) 0)
        no-repeat 100% 0 / 2em 2em,
    linear-gradient(-135deg, transparent 2em, #58a 0);
```

Как подтверждает рис. 4.27, результат не совсем такой, как ожидалось. Почему же размер не совпадает? Ведь высота обоих градиентов равна `2em`!

Причина заключается в том (как мы уже обсуждали в **секрете «Срезанные углы»**), что размер угла, равный `2em`, в случае второго градиента относится к позиции границы перехода цвета и, следовательно, отмеряется по градиентной линии, то есть по диагонали. С другой стороны, `2em` для `background-size` — это ширина и высота фоновой плитки, которые измеряются соответственно по горизонтали и по вертикали.

Для того чтобы совместить эти два градиента, необходимо сделать одно из следующего, в зависимости от того, какой размер вы хотите сохранить:

- чтобы диагональ все так же была равна `2em`, мы можем умножить `background-size` на $\sqrt{2}$;
- чтобы сохранить ширину и высоту, равные `2em`, можно разделить позицию границы перехода цвета нашего градиента для срезанного угла на $\sqrt{2}$.

Так как `background-size` повторяется дважды, а большинство других измерений в CSS делается **не** по диагонали, второй вариант обычно предпочтительнее.

Рис. 4.26. Наш второй градиент для треугольника, представляющего загнутый уголок, отдельно от всего остального. Текст показан светло-серым цветом вместо белого, для того чтобы вы могли видеть его

Рис. 4.27. Объединение двух градиентов не дает желаемого результата

«Единственный способ избавиться от искушения — это поддаться ему».
Оскар Уайлд,
«Портрет Дориана Грея»

Рис. 4.28. Наш загнутый уголок наконец-то оказывается на своем месте, когда мы меняем позицию границы перехода цвета для синего градиента

Позиция границы перехода цвета примет значение $2/\sqrt{2} = \sqrt{2} \approx 1{,}414213562$, что мы округлим до `1.5em`:

```
background: #58a; /* Резервное решение */
background:
    linear-gradient(to left bottom,
        transparent 50%, rgba(0,0,0,.4) 0)
        no-repeat 100% 0 / 2em 2em,
    linear-gradient(-135deg,
        transparent 1.5em, #58a 0);
```

Как видно на рис. 4.28, это наконец-то позволяет получить приятный глазу, гибкий и минималистичный загнутый уголок.

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**folded-corner**

Убедитесь, что ширина забивки по меньшей мере равна величине угла, иначе текст будет наползать на угол (ведь это всего лишь фон), разрушая иллюзию загнутого уголка страницы.

Решение для других углов

В реальности уголки страниц редко загибаются под углом 45°. Если нам требуется нечто более реалистичное, то мы могли бы попробовать указать другое значение угла, например `-150deg` для угла 30°. Но это всего лишь изменит угол наклона срезанного угла, а треугольник, представляющий загнутую часть страницы, останется на своем месте, полностью исказив эффект, как показано на рис. 4.29. Однако скорректировать размеры этого треугольника не так просто, как может показаться. Его размер определяется не углом, а шириной и высотой. Как же нам понять, какую ширину и высоту указать для нужного эффекта? Что ж, пришло время — о нет! — тригонометрии!

«Единственный способ избавиться от искушения — это поддаться ему».
Оскар Уайлд,
«Портрет Дориана Грея»

Рис. 4.29. Изменение угла срезки приводит к разрушению эффекта

В настоящее время код выглядит так:

```
background: #58a; /* Резервное решение */
background:
    linear-gradient(to left bottom,
        transparent 50%, rgba(0,0,0,.4) 0)
        no-repeat 100% 0 / 2em 2em,
    linear-gradient(-150deg,
        transparent 1.5em, #58a 0);
```

Как показано на рис. 4.30, по сути, нам нужно вычислить длину гипотенузы прямоугольного треугольника с углами 30, 60 и 90 градусов, при условии, что нам известна длина одного из катетов. Тригонометрический круг на рис. 4.31 напоминает, что если нам известны **углы** и **длина одной из сторон прямоугольного треугольника**, то мы можем вычислить длины остальных двух сторон, используя синусы, косинусы и теорему Пифагора. Из курса математики (или сверившись с калькулятором) мы знаем, что $\cos 30° = \frac{\sqrt{3}}{2}$, а $\sin 30° = \frac{1}{2}$. Также тригонометрический круг подсказывает, что в нашем случае $\sin 30° = \frac{1{,}5}{x}$, а $\cos 30° = \frac{1{,}5}{y}$. Следовательно:

Рассмотрим прямоугольный треугольник с углами 90°, 30° и 60°.

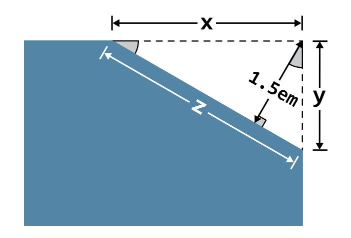


Рис. 4.30. Наш срезанный угол в масштабированном представлении (углы, помеченные серым цветом, равны 30°)

$$\frac{1}{2} = \frac{1{,}5}{x} \Rightarrow x = 2 \times 1{,}5 \Rightarrow x = 3$$

$$\frac{\sqrt{3}}{2} = \frac{1{,}5}{y} \Rightarrow y = \frac{2 \times 1{,}5}{\sqrt{3}} \Rightarrow y = \sqrt{3} \approx 1{,}732050808.$$

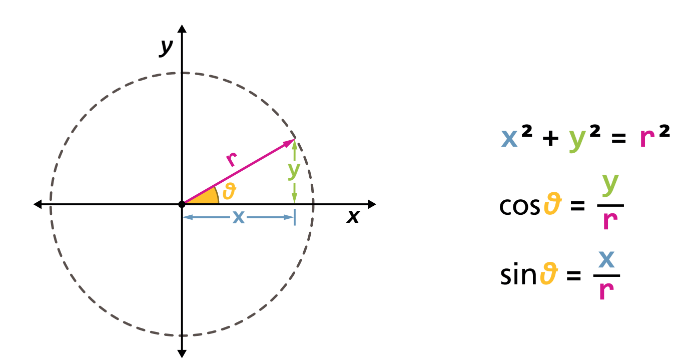


Рис. 4.31. Синусы и косинусы помогают вычислять длины катетов прямоугольных треугольников, когда известны значения углов и длина гипотенузы

Теперь мы можем с помощью теоремы Пифагора вычислить значение z:

$$z = \sqrt{x^2 + y^2} = \sqrt{\sqrt{3}^2 + 3^2} + \sqrt{3 + 9} = \sqrt{12} = 2\sqrt{3}.$$

А теперь соответствующим образом изменим размер треугольника:

```
background: #58a; /* Резервное решение */
background:
    linear-gradient(to left bottom,
        transparent 50%, rgba(0,0,0,.4) 0)
        no-repeat 100% 0 / 3em 1.73em,
    linear-gradient(-150deg,
        transparent 1.5em, #58a 0);
```

«Единственный способ избавиться от искушения — это поддаться ему».
Оскар Уайлд,
«Портрет Дориана Грея»

Рис. 4.32. Хотя мы добились желаемого результата, на практике он выглядит еще менее реалистичным, чем раньше

Сейчас угол выглядит как на рис. 4.32. Вы видите, что край треугольника **совпадает со срезанным углом**, но результат выглядит еще **менее реалистичным**! Хотя конкретная причина становится понятна не сразу, мы в своей жизни видели достаточно загнутых уголков и мгновенно понимаем, что что-то здесь не так. Помочь своему сознанию разобраться в происходящем можно, **попробовав загнуть уголок настоящего листа бумаги** примерно под таким же углом. Вы быстро поймете, что **не существует способа** загнуть его так, чтобы он хотя бы приблизительно напоминал показанное на рис. 4.32.



Рис. 4.33. Аналоговая версия эффекта загнутого уголка (за предоставленный шикарно украшенный лист бумаги благодарю Леони и Фиби Веру)

Как доказывает настоящий загнутый уголок из реальной жизни, например показанный на рис. 4.33, нужный нам треугольник должен быть слегка повернут и обладать теми же габаритными размерами, что и треугольник, «отрезанный» от угла элемента. Поскольку поворачивать фоны мы не можем, настало время применить эффект к псевдоэлементу:

```
.note {
    position: relative;
    background: #58a; /* Резервное решение */
    background:
        linear-gradient(-150deg,
            transparent 1.5em, #58a 0);
}
.note::before {
    content: '';
    position: absolute;
    top: 0; right: 0;
    background: linear-gradient(to left bottom,
        transparent 50%, rgba(0,0,0,.4) 0)
        100% 0 no-repeat;
    width: 3em;
    height: 1.73em;
}
```

Пока что мы всего лишь воссоздали эффект, показанный на рис. 4.32, с использованием псевдоэлементов. Следующим шагом будет изменение ориентации существующего треугольника путем **замены** `width` **на** `height` **и наоборот**, чтобы он **зеркально отражал срезанный угол**, а не дополнял его. Затем мы повернем его на 30° ((90° – 30°) – 30°) против часовой стрелки, для того чтобы его **гипотенуза была параллельна нашему срезанному углу**:

```
.note::before {
    content: '';
    position: absolute;
    top: 0; right: 0;
    background: linear-gradient(to left bottom,
        transparent 50%, rgba(0,0,0,.4) 0)
        100% 0 no-repeat;
    width: 1.73em;
    height: 3em;
    transform: rotate(-30deg);
}
```

«Единственный способ избавиться от искушения — это поддаться ему». Оскар Уайлд, «Портрет Дориана Грея»

Рис. 4.34. Мы приближаемся к желаемому результату, но нам нужно повернуть треугольник

Как выглядит наша записка после этих изменений, вы видите на рис. 4.34. Мы уже почти достигли желаемого эффекта, осталось лишь сдвинуть треугольник, для того чтобы гипотенузы двух треугольников (темного и представляющего срезанный угол) совпали. Судя по тому, как обстоят дела сейчас, нам нужно сдвинуть треугольник и по горизонтали, и по вертикали, и понять, какие именно действия необходимо произвести, не так-то просто. Упростить себе задачу можно, установив для свойства `transform-origin` значение `bottom right`, чтобы нижний **правый угол треугольника стал центром вращения** и, таким образом, был зафиксирован на одном месте:

```
.note::before {
    /* [Остальные определения стилей] */
    transform: rotate(-30deg);
    transform-origin: bottom right;
}
```

«Единственный способ избавиться от искушения — это поддаться ему». Оскар Уайлд, «Портрет Дориана Грея»

Рис. 4.35. Добавление transform-origin: bottom right; упрощает ситуацию: теперь нам нужно только сместить треугольник по вертикали

Как видно на рис. 4.35, теперь нам осталось только сдвинуть треугольник вертикально вверх. Определить точную величину сдвига нам снова поможет геометрия. Схема на рис. 4.36 помогает увидеть, что требуемое вертикальное смещение для нашего треугольника равно $x - y = 3 - \sqrt{3} \approx 1{,}267949192$, что можно округлить до `1.3em`:

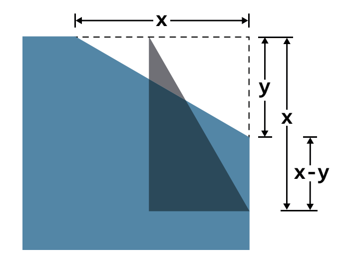


Рис. 4.36. Вычислить, на какую величину необходимо сдвинуть треугольник, не так сложно, как кажется

«Единственный способ избавиться от искушения — это поддаться ему». Оскар Уайлд, «Портрет Дориана Грея»

Рис. 4.37. Наши треугольники наконец выровнены, и их стороны совпадают

```
.note::before {
    /* [Остальные определения стилей] */
    transform: translateY(-1.3em) rotate(-30deg);
    transform-origin: bottom right;
}
```

Пример визуализации на рис. 4.37 подтверждает, что мы наконец добились желаемого эффекта. Уф-ф-ф, это было нелегко! Кроме того, поскольку теперь наш треугольник генерируется с помощью псевдоэлементов, мы можем сделать его **еще более реалистичным**, добавив скругленные углы (настоящие) градиенты и тени `box-shadow`! Финальная версия кода выглядит так:

```
.note {
    position: relative;
    background: #58a; /* Fallback */
    background:
        linear-gradient(-150deg,
            transparent 1.5em, #58a 0);
    border-radius: .5em;
}
.note::before {
    content: '';
    position: absolute;
    top: 0; right: 0;
    background: linear-gradient(to left bottom,
        transparent 50%, rgba(0,0,0,.2) 0, rgba(0,0,0,.4))
        100% 0 no-repeat;
    width: 1.73em;
    height: 3em;
    transform: translateY(-1.3em) rotate(-30deg);
    transform-origin: bottom right;
    border-bottom-left-radius: inherit;
    box-shadow: -.2em .2em .3em -.1em rgba(0,0,0,.15);
}
```

Насладиться плодами нашего труда можно, взглянув на рис. 4.38.

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**folded-corner-realistic**

Эффект смотрится замечательно, но насколько он соответствует принципам DRY? Давайте подумаем, какие правки и вариации могут часто требоваться в дизайнах, использующих данных эффект:

- достаточно только **одной правки** для изменения **габаритных размеров элемента и других длин** (забивки и т. п.);
- достаточно только **двух правок** (одной, если не брать в расчет резервное решение) для изменения **цвета фона**;
- необходимо **четыре правки и несколько нетривиальных вычислений**, чтобы изменить размер загнутого уголка;
- необходимо **пять правок и несколько еще менее тривиальных вычислений**, чтобы изменить угол, под которым загнут уголок страницы.

Два последних пункта никуда не годятся. Возможно, настало время прибегнуть к помощи препроцессорной примеси:

SCSS

```
@mixin folded-corner($background, $size,
                     $angle: 30deg) {
position: relative;
background: $background; /* Резервное решение */
background:
    linear-gradient($angle - 180deg,
        transparent $size, $background 0);
border-radius: .5em;

$x: $size / sin($angle);
$y: $size / cos($angle);

&::before {
    content: '';
    position: absolute;
    top: 0; right: 0;
    background: linear-gradient(to left bottom,
        transparent 50%, rgba(0,0,0,.2) 0,
        rgba(0,0,0,.4)) 100% 0 no-repeat;
    width: $y; height: $x;
    transform: translateY($y - $x)
        rotate(2*$angle - 90deg);
```



Убедитесь, что трансформация `translateY()` определена **перед** вращением — в противном случае наш треугольник будет двигаться относительно своего угла 30°, так как каждая трансформация также преобразует всю систему координат элемента, а не только сам элемент как таковой!

«Единственный способ избавиться от искушения — это поддаться ему». Оскар Уайлд, «Портрет Дориана Грея»

Рис. 4.38. Еще несколько эффектов — и наш загнутый уголок оживает



На момент написания этой главы SCSS в исходном формате не поддерживает тригонометрические функции. Для того чтобы обеспечить такую поддержку, вы можете воспользоваться каркасом **Compass** (http://compass-style.org) или одной из других библиотек. Вы можете даже писать их самостоятельно, используя расширения функций Тейлора! **LESS**, с другой стороны, включает их по умолчанию с самого начала.

```
    transform-origin: bottom right;
    border-bottom-left-radius: inherit;
    box-shadow: -.2em .2em .3em -.1em rgba(0,0,0,.2);
}
}

/* Использовать как... */
.note {
    @include folded-corner(#58a, 2em, 40deg);
}
```

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**folded-corner-mixin**

СВЯЗАННЫЕ СПЕЦИФИКАЦИИ

CSS Backgrounds & Borders: http://w3.org/TR/css-backgrounds

CSS Image Values: http://w3.org/TR/css-images

CSS Transforms: http://w3.org/TR/css-transforms

5

Оформление текста

20 Расстановка переносов

Проблема

Дизайнеры просто обожают выравнивание текста по ширине. Взглянув на шикарно оформленный журнал или книгу, вы увидите, что этот прием используется повсеместно. Однако в Сети выравнивание по ширине встречается гораздо реже, особенно в работах опытных дизайнеров. Почему же так происходит, учитывая, что `text-align: justify;` присутствует в нашем арсенале со времен CSS 1?

Причина станет очевидной, если вы присмотритесь к «коридорам» на рис. 5.1. Они создают увеличенные межсловные интервалы, призванные выровнять текст слева и справа. Это не только ужасно выглядит, но и **ухудшает читабельность**. В печатном дизайне **выравнивание по ширине** всегда сочетается **с расстановкой переносов**. Поскольку слова при этом разбиваются на слоги, дополнительных пробелов почти не требуется, и текст в результате выглядит намного естественнее.

“The only way to get rid of a temptation is to yield to it.”

Рис. 5.1. Эффект, создаваемый стандартной функцией CSS выравнивания текста по ширине

До недавнего времени включить переносы на веб-странице было настолько сложно, что **решение оказывалось хуже самой проблемы**. Типичный сценарий предполагал использование кода на стороне сервера, код JavaScript, интерактивные генераторы, а также руки разработчика и безграничное терпение, чтобы расставить мягкие переносы (``), подсказывая браузеру, в каком месте каждое слово **может** быть разорвано. Обычно такие трудозатраты себя не оправдывали, и дизайнеры прибегали к другому способу выравнивания текста.

Решение

В CSS Text Level 3 появилось новое свойство: `hyphens`. Оно способно принимать три значения: `none`, `manual` и `auto`. Первоначальное значение равно `manual`, и оно соответствует существующему в настоящее время поведению: слова всегда можно разбить на слоги вручную, используя мягкие переносы. Очевидно, что `hyphens: none;` отключает такое поведение, но поистине волшебных результатов позволяет достигать вот эта очень простая строка CSS-кода:

“The only way to get rid of a temptation is to yield to it.”

Рис. 5.2. Результат применения атрибута `hyphens: auto`

```
hyphens: auto;
```

Это все, что нам нужно. Результат вы видите на рис. 5.2. Разумеется, чтобы это работало, необходимо объявить язык посредством HTML-атрибута `lang`, но хороший разработчик должен делать это в любом случае, независимо от переносов.

Если вам требуется более высокая степень контроля над расстановкой переносов (например, в коротком вступительном тексте), **вы все так же можете помочь браузеру, добавив несколько мягких переносов** (``). Свойство `hyphens` **отдает им приоритет** и только после этого начинает работать, выясняя, **где еще возможно разбить слова на слоги**.

ЗАНИМАТЕЛЬНАЯ СТРАНИЧКА. КАК РАБОТАЕТ ОБТЕКАНИЕ ТЕКСТОМ

Как это часто бывает в компьютерных науках, обтекание текстом кажется чем-то простым и прямолинейным, но в действительности это не так. Существует множество алгоритмов, выполняющих данную функцию, среди которых самые популярные — жадный алгоритм (greedy algorithm) и алгоритм Кнута — Пласса (Knuth — Plass algorithm). *Жадный алгоритм* работает, анализируя одну строку текста за раз и заполняя ее как можно большим количеством слов (или слогов, если используется расстановка переносов). Переход на следующую строку выполняется, когда алгоритм встречает первое слово/слог, которое не умещается в текущую строку.

Алгоритм Кнута — Пласса, названный в честь разработавших его инженеров, намного изощреннее. Он рассматривает весь текст целиком и выдает намного более приятные с эстетической точки зрения результаты, хотя обработка текста, конечно же, занимает куда больше времени.

В большинстве текстовых редакторов используется алгоритм Кнута — Пласса. Однако в браузерах в настоящее время по причинам, связанным с производительностью, реализован жадный алгоритм, поэтому результаты выравнивания текста по ширине не слишком хороши.

Резервное решение для расстановки переносов средствами CSS выглядит довольно изящно. Если свойство `hyphens` не поддерживается, то вы получаете выровненный по ширине текст, который выглядит как на рис. 5.1. Конечно, его не так приятно читать, и выглядит он неэстетично, но все же такой вариант допустим.

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**hyphenation**

СВЯЗАННЫЕ СПЕЦИФИКАЦИИ

CSS Text: http://w3.org/TR/css-text

CSS Text Level 4: http://dev.w3.org/csswg/css-text-4

БУДУЩЕЕ.
КОНТРОЛЬ НАД РАССТАНОВКОЙ ПЕРЕНОСОВ

Если вы вышли из дизайнерской среды, возможно, вас коробит такой подход к расстановке переносов, когда их можно только включить или выключить, но нельзя контролировать, как именно слова будут разделены на слоги.

Тогда вас порадует новость, что в будущем у нас появятся намного более точные средства управления расстановкой переносов — несколько связанных с этим свойств уже запланированы для CSS Text Level 4 (http://dev.w3.org/csswg/css-text-4), в частности:

- `hyphenate-limit-lines`
- `hyphenate-limit-chars`
- `hyphenate-limit-zone`
- `hyphenate-limit-last`
- `hyphenate-character`

21 Вставка разрыва строки

Проблема

Необходимость разрывать строки средствами CSS обычно возникает при использовании списков определений (рис. 5.3), но также в некоторых других случаях. Чаще всего мы используем списки определений, потому что хотим быть добропорядочными кибергражданами и создавать правильную семантическую разметку, даже если **визуальный** результат, который нам требуется, — это всего лишь **несколько строк с парами из имени и значения**. Рассмотрим такую разметку:

Name: **Lea Verou**
Email: **lea@verou.me**
Location: **Earth**

Рис. 5.3. Список определений, где в каждой строке находится пара из имени и значения

HTML

```
<dl>
    <dt>Name:</dt>
    <dd>Lea Verou</dd>

    <dt>Email:</dt>
    <dd>lea@verou.me</dd>

    <dt>Location:</dt>
    a<dd>Earth</dd>
</dl>
```

Name:
Lea Verou
Email:
lea@verou.me
Location:
Earth

Рис. 5.4. Стилизация по умолчанию для нашего списка определений

Ожидаемый визуальный результат — простая стилизация, показанная на рис. 5.3. На первом шаге мы чаще всего применяем пару простых приемов CSS, например:

```
dd {
    margin: 0;
    font-weight: bold;
}
```

Name: **Lea Verou** Email: **lea@verou.me** Location: **Earth**

Рис. 5.5. display: inline разрывает строки еще хуже

Однако поскольку <dt> и <dd> — это блочные элементы, в результате у нас получается нечто более напоминающее рис. 5.4, когда каждое имя и каждое значение отображаются на отдельной строке. Последующие попытки обычно включают тестирование разных значений свойства display для элемента <dt>, <dd> или обоих, вплоть до абсолютно произвольных, по мере того как мы приходим в полное отчаяние. Но результат при этом чаще всего оказывается похожим на рис. 5.5.

Прежде чем рвать на себе волосы, проклиная всех богов CSS, или прощаться с идеей разделения понятий и переходить к модифицированию разметки, нужно все же оценить, нет ли способа сохранить ясность ума и высокие стандарты кодирования.

Решение

По сути нам нужно только добавить переносы строк в конце каждого <dd>. Если мы ничего не имеем против презентационной разметки, то можно сделать это с помощью старых добрых элементов
. Скажем, так:

HTML

```
<!-- Каждый раз, когда вы делаете это, где-то умирает котенок -->
<dt>Name:</dt>
<dd>Lea Verou<br /></dd>
...
```

Затем мы применили бы display:inline; ко всем <dt> и <dd> и заявили, что дело сделано. Разумеется, это не только плохая практика с точки зрения поддержки; при использовании данного подхода код несоразмерно раздувается. Если бы мы могли использовать генерируемое содержимое для добавления переносов строки, работающих аналогично элементам
, это сразу решило бы нашу проблему! Но мы этого сделать не можем... *Или все же можем?*

В действительности существует символ Unicode, соответствующий переносу строки: 0x000A. В CSS-коде мы должны записывать его как "\000A" или, еще проще, "\A". Его можно использовать в качестве содержимого нашего псевдоэлемента ::after, чтобы он автоматически добавлялся в конце каждого <dd>, например так:

```
dd::after {
    content: "\A";
}
```

Вроде бы это должно сработать, но если мы попытаемся применить этот код, результат нас разочарует: по сравнению с рис. 5.5 ничего не изменится. Но это не означает, что мы идем по неверному пути, просто **мы кое-что забыли**. На самом деле то, что мы делаем с помощью этого CSS-кода, эквивалентно добавлению переносов строки в нашей HTML-разметке прямо перед закрывающимися тегами `</dd>`. Помните, что происходит с переносами строк в коде HTML? По умолчанию они **схлопываются** вместе с остальным пустым пространством. Чаще всего это как раз то, что нам нужно, так как в противном случае нам пришлось бы форматировать всю HTML-страницу целиком как одну строку. Однако иногда **пустое пространство и переносы строки нужно сохранять**, как, например, в блоках с примерами кода. И что же мы обычно делаем в таких ситуациях? Мы применяем `white-space: pre;`. То же самое можно сделать и в нашем примере, но только для генерируемых переносов строк.

Технически `0x000A` соответствует символу Line Feed («Перевод строки»), который мы получаем в JavaScript, когда используем `"\n"`. Есть также символ Carriage Return («Возврат каретки», `"\r"` в JS, `"\D"` в CSS), но в современных браузерах он не требуется.

Все, что у нас есть, — это один символ переноса строки, поэтому нас не особо волнует, сохранится пустое пространство или нет (оно все равно отсутствует). Следовательно, нам подойдет любое значение `pre` (`pre`, `pre-line`, `pre-wrap`). Я рекомендую использовать `pre`, так как оно поддерживается наибольшим количеством браузеров. Давайте соберем все вместе:

```
dt, dd { display: inline; }

dd {
    margin: 0;
    font-weight: bold;
}

dd::after {
    content: "\A";
    white-space: pre;
}
```

Протестировав этот код, вы увидите, что он действительно работает и обеспечивает результат, в точности соответствующий представленному на рис. 5.3! Но насколько это решение гибкое? Предположим, мы хотим добавить второй адрес электронной почты к записям пользователей, которые содержатся в нашем списке определений:

HTML

```
...
<dt>Email:</dt>
<dd>lea@verou.me</dd>
<dd>leaverou@mit.edu</dd>
...
```

Name: **Lea Verou**
Email: **lea@verou.me**
leaverou@mit.edu
Location: **Earth**

Рис. 5.6. Наше решение ломается, когда используется несколько элементов `<dd>`

Теперь результат выглядит, как показано на рис. 5.6, и это действительно странно. Поскольку **перенос строки добавляется после каждого** `<dd>`, каждое значение выводится на отдельной строке, даже если необходимости переносить его на новую строку нет. Было бы намного лучше, если бы множественные значения разделялись запятыми, но оставались на одной строке (при условии, что там достаточно места).

В идеальном случае нам хотелось бы выбирать только последние `<dd>` перед `<dt>` и добавлять переносы строк только для них, но не для всех элементов `<dd>`. Однако в своем текущем состоянии селекторы CSS не обеспечивают такой точности, потому что не способны заглядывать вперед и проверять элементы после субъекта в DOM-дереве. Необходимо придумать другой способ. Одна из идей — попробовать добавлять переносы строк **перед** `<dt>`, а не **после** `<dd>`:

```
dt::before {
    content: '\A';
    white-space: pre;
}
```

Однако это приводит к появлению пустой первой строки, поскольку селектор применяется также и к первому `<dt>`. Для того чтобы справиться с этим, можно попытаться использовать любой из следующих селекторов вместо `dt`:

- `dt:not(:first-child)`
- `dt ~ dt`
- `dd + dt`

Мы выберем последний вариант, так как он будет работать, в том числе и в сценарии, когда для одного и того же значения определено несколько элементов `<dt>`, в отличие от первых двух селекторов, которые в таких условиях ломаются. Также нам необходимо как-то разделять множественные элементы `<dd>`, если использование в качестве разделителя обыкновенного пробела нас не удовлетворяет (это допустимо в одних случаях, но дает плохие результаты в других). В идеальной ситуации нам бы хотелось иметь возможность сказать браузеру: «Добавляй запятую после каждого `<dd>`, предшествующего другому `<dd>`», но

опять же, современные селекторы CSS не настолько хороши. Таким образом, нам придется добавлять запятую **перед** каждым <dd>, следующим за другим <dd>. Результирующий CSS-код представлен далее (а результат вы можете увидеть на рис. 5.7):

```
dd + dt::before {
    content: '\A';
    white-space: pre;
}

dd + dd::before {
    content: ', ';
    font-weight: normal;
}
```

Name: **Lea Verou**

Email: **lea@verou.me, leaverou@mit.edu**

Location: **Earth**

Рис. 5.7. Итоговый результат

Помните, что если ваша разметка включает (незакомментированное) пустое пространство между множественными последовательными элементами <dd>, то **перед запятой появится пробел**. Есть несколько способов справиться с этим недостатком, но ни один из них не идеален. Например, **поля отрицательного размера**:

```
dd + dd::before {
    content: ', ';
    margin-left: -.25em;
    font-weight: normal;
}
```

Это решение будет работать, но оно довольно хрупкое. В случае отображения содержимого **на другом фоне** и **с другими метриками** это пространство может оказаться шире или уже 0.25em, и тогда результат будет выглядеть странно. Однако с большинством шрифтов разница пренебрежимо мала.

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**line-breaks**

22 Полосатая заливка строк текста

ПРЕДВАРИТЕЛЬНЫЕ ТРЕБОВАНИЯ

Градиенты CSS, свойство `background-size`, секрет «Фон в полоску», секрет «Гибкое позиционирование фона»

Проблема

Когда несколько лет назад мы впервые получили псевдоклассы `:nth-child()`/`:nth-of-type()`, одним из наиболее распространенных вариантов их применения стали **таблицы с полосатой заливкой «зеброй»** (рис. 5.8). То, что раньше требовало написания кода на серверной стороне, сценариев на клиентской стороне или утомительного ручного кодирования, теперь может быть реализовано всего лишь несколькими строками кода:

```
tr:nth-child(even) {
    background: rgba(0,0,0,.2);
}
```

Однако когда дело касается применения того же эффекта к строкам текста, а не к строкам таблицы, мы все так же бессильны. Этот прием особенно удобен **для оформления фрагментов кода**, так как позволяет делать код более читабельным. Многие разработчики прибегают к помощи JavaScript, оборачивая каждую строку в собственный `<div>`, для того чтобы все так же иметь возможность использовать технику с `:nth-child()`, часто абстрагируясь от этого безобразия с помощью функций подсветки синтаксиса. Но это не только субоптимально с точки зрения чистоты кода (код

Многие разработчики отправляли запросы на добавление псевдокласса `:nth-line()` в рабочую группу CSS, но эти просьбы были отвергнуты по причинам, связанным с производительностью.

JS не должен быть связан со стилизацией); **слишком большое количество DOM-элементов может замедлить работу страницы**, кроме того, это **в любом случае хрупкое решение** (что произойдет, если вы увеличите размер шрифта и потребуется дать текст с обтеканием?). Существует ли лучший способ?

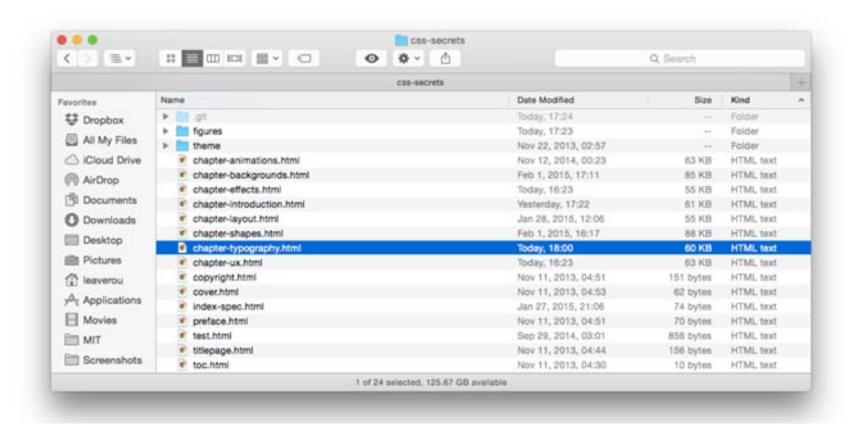


Рис. 5.8. Таблицы с полосатой заливкой «зеброй» всегда были популярны как в дизайне пользовательских интерфейсов (так оформлен список файлов в Mac OS X), так и в печатном дизайне, поскольку чередование фонового цвета помогает отслеживать взглядом содержимое длинной строки

Решение

Вместо того чтобы зацикливаться на добавлении темного фона к элементам, представляющим строки, давайте взглянем на проблему под другим углом. Почему бы не применить свойство, создающее фоновое изображение, **ко всему элементу** и не **определить чередующиеся цвета на этом фоновом изображении**? На первый взгляд идея кажется ужасной, но вспомните, что **мы можем генерировать фоны непосредственно с помощью CSS-кода**, используя градиенты CSS, а размеры указывать в единицах `em`, заставляя их **автоматически адаптироваться к изменениям значения** `font-size`.

```
while (true) {
  var d = new Date();
  if (d.getDate()==1 &&
      d.getMonth()==3) {
    alert("TROLOLOL");
  }
}
```

Рис. 5.9. Фрагмент кода без чередования заливки, на старом добром фоне сплошного цвета

Давайте попробуем развить эту идею, чтобы поместить фрагмент кода с рис. 5.9 на фон с чередующимися цветными полосами. Для начала нам нужно создать горизонтальные полосы, применив технику из **секрета «Фон в полоску»**. Значение `background-size` должно **в два раза превышать**

```
while (true) {
  var d = new Date();
  if (d.getDate()==1 &&
      d.getMonth()==3) {
    alert("TROLOLOL");
  }
}
```

Рис. 5.10. Наша первая попытка поместить фрагмент кода на фон с чередующимися цветными полосами

```
while (true) {
  var d = new Date();
  if (d.getDate()==1 &&
      d.getMonth()==3) {
    alert("TROLOLOL");
  }
}
```

Рис. 5.11. Готовый результат

Почему мы не воспользовались простым сокращением `background` для всех значений, связанных с фоном? Потому что тогда нам потребовалось бы отдельное объявление для резервного решения, предназначенного для старых браузеров. То есть нам пришлось бы дважды упоминать значение `beige`, что иллюстрирует принцип WET, но никак не DRY.

значение `line-height`, так как **каждая плитка градиента охватывает две строки**. Первая попытка создать нужный код выглядит примерно так:

```
padding: .5em;
line-height: 1.5;
background: beige;
background-image: linear-gradient(rgba(0,0,0,.2)
50%, transparent 0);
background-size: auto 3em;
```

Как видно на рис. 5.10, результат **очень близок к тому, чего мы хотим добиться**. Мы можем даже менять размер шрифта, и полосы будут соответствующим образом сжиматься и расширяться! Однако в глаза сразу бросается серьезная проблема: полосы **не выровнены** по строкам кода, что вроде как сводит на нет все наши усилия. Почему же так происходит?

Если вы внимательно посмотрите на рис. 5.10, то заметите, что первая полоса начинается вверху контейнера — ожидаемое поведение фонового изображения. Но **наш код начинается ниже**, так как он смотрелся бы уродливо, будучи прижатым к верхней кромке контейнера. Как вы видите, мы добавили забивку шириной `.5em`, и это как раз то смещение, которое отделяет наши полосы от желаемой позиции.

Одним из вариантов решения данной проблемы было бы использование `background-position`, чтобы подвинуть полосы на `.5em` вниз. Однако если позднее мы решим изменить величину забивки, то нам также потребуется скорректировать позицию фона, что не соответствует принципам DRY. Можно ли сделать так, чтобы **фон автоматически подстраивался под величину забивки**?

Помните свойство `background-origin` из **секрета «Гибкое позиционирование фона»**? Это как раз то, что нам нужно: способ приказать браузеру использовать в качестве точки отсчета для вычисления значения `background-position` кромку поля содержимого, а не кромку поля забивки, которая используется по умолчанию. Давайте добавим это в наш рецепт:

```
padding: .5em;
line-height: 1.5;
background: beige;
background-size: auto 3em;
background-origin: content-box;
background-image: linear-gradient(rgba(0,0,0,.2) 50%, transparent 0);
```

Как видно на рис. 5.11, это именно то, чего мы ожидали от полосатой заливки строк! Так как для полос мы использовали полупрозрачные цвета, мы можем даже изменить цвет основного фона, и полосы все так же будут видны. По сути, это решение настолько гибкое, что **единственный способ сломать его**[1] **— изменить значение `line-height`, не скорректировав соответствующим образом `background-size`.**

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**zebra-lines**

СВЯЗАННЫЕ СПЕЦИФИКАЦИИ

CSS Backgrounds & Borders: http://w3.org/TR/css-backgrounds

CSS Image Values: http://w3.org/TR/css-images

---

[1] Здесь мы предполагаем, что имеем дело с фрагментами кода. В общем случае решение может сломаться из-за строковых элементов, увеличивающих высоту строки, например изображений или строкового содержимого с бо́льшим значением `font-size`.

23 Корректировка величины табуляции

Проблема

С веб-страницами, содержащими большое количество кода, такими как страницы документации или учебников, связаны собственные уникальные сложности стилизации. К элементам `<pre>` и `<code>`, с помощью которых мы оформляем код, применяется стилизация по умолчанию, определяемая пользовательским агентом, например такая:

```
pre, code {
  font-family: monospace;
}

pre {
  display: block;
  margin: 1em 0;
  white-space: pre;
}
```

```
while (true) {
        var d = new Date();
        if (d.getDate()==1
            d.getMonth()==3
                alert("TROL
        }
}
```

Рис. 5.12. Так код выглядит с величиной табуляции по умолчанию, равной восьми символам

Однако этого определенно недостаточно для того, чтобы учесть все уникальные трудности отображения кода. Одна из самых серьезных проблем заключается в том, что хотя **табуляция идеально подходит для создания отступов в коде**, очень часто ее стараются избегать, так как место, выделяемое браузером для отображения табуляции, по ширине равно целым восьми (!) символам. Посмотрите на рис. 5.12 — как плохо выглядят такие большие отступы и как бездарно расходуется место на экране: наш код даже не поместился в свой контейнер!

Вы поморщились при упоминании табуляции как средства создания отступов в коде? Это не входит в список тем, рассматриваемых в данной книге, но мои доводы представлены на странице http://lea.verou.me/2012/01/whytabs-are-clearly-superior.

Решение

К счастью, в **CSS Text Level 3** у нас появилось новое свойство CSS для управления этой величиной: `tab-size`. В качестве значения оно принимает **число** (количество символов) или **длину** (что редко бывает полезным). Чаще всего мы задаем значение `4` (то есть ширина четырех символов) или `2` — последняя тенденция в оформлении отступов строк кода:

```
pre {
    tab-size: 2;
}
```

Как видно на рис. 5.13, код теперь читать намного проще. Вы могли бы даже установить значение `tab-size`, равное `0`, для того чтобы полностью отключить табуляцию, но это редко (или вообще никогда) дает хорошие результаты, что подтверждает рис. 5.14. Если свойство не поддерживается, то ничего не ломается — мы просто получаем ужасно широкую табуляцию, с которой нам и так приходилось мириться все эти годы.

```
while (true) {
  var d = new Date();
  if (d.getDate()==1 &&
      d.getMonth()==3) {
    alert("TROLOLOL");
  }
}
```

Рис. 5.13. Тот же код, что и на рис. 5.12, здесь отображается с табуляцией, равной ширине двух символов

```
while (true) {
var d = new Date();
if (d.getDate()==1 &&
    d.getMonth()==3) {
alert("TROLOLOL");
 }
}
```

Рис. 5.14. Код отображается с нулевой табуляцией, вследствие чего все отступы, созданные табуляцией, исчезают. Не делайте так!

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**tab-size**

СВЯЗАННЫЕ СПЕЦИФИКАЦИИ

CSS Text: http://w3.org/TR/css-text

24 Лигатуры

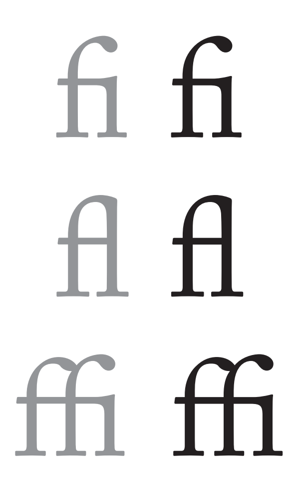


Рис. 5.15. Распространенные лигатуры, которые можно обнаружить в большинстве шрифтов с засечками

Проблема

Так же как и люди, не все глифы хорошо уживаются вместе. Взять, например, символы **f** и **i** в большинстве шрифтов с засечками. Точка, принадлежащая символу **i**, часто налезает на верхний выносной элемент символа **f**, из-за чего эта пара выглядит некрасиво (первый пример на рис. 5.15).

Чтобы устранить это недоразумение, в дизайн шрифтов часто включают **дополнительные глифы**, называемые *лигатурами*. Это индивидуально разработанные пары и тройки глифов, которые программа набора текста автоматически подставляет, когда соответствующие символы находятся друг рядом с другом. Например, на рис. 5.15 вы видите несколько распространенных лигатур. Обратите внимание, насколько лучше они смотрятся, чем просто находящиеся рядом эквивалентные глифы.

Также существуют так называемые *дискретные лигатуры* (рис. 5.16), которые разрабатываются ради стилистической альтернативы, а не потому, что с соответствующими символами связаны какие-то проблемы отображения, когда те стоят вплотную друг к другу.

Однако в браузерах дискретные лигатуры никогда по умолчанию не используются (и это правильно), а зачастую не используются и обыкновенные лигатуры (а это уже ошибка). В действительности до недавнего времени единственным способом задействовать любую лигатуру было добавление в код эквивалентного

символа Unicode: например, `ﬁ` для лигатуры **fi**. Но этот метод создает больше проблем, чем решает:

- очевидно, что разметку становится трудно читать и еще сложнее писать (попробуйте догадаться, что за слово кроется в шифре `deﬁne`!);
- если текущий шрифт не включает символ для данной лигатуры, то результат начинает смахивать на анонимки из вырезанных газетных букв (рис. 5.17);
- не для каждой лигатуры существует эквивалентный стандартизированный символ Unicode. Например, лигатуре **ct** не соответствует никакой символ Unicode, и в любых шрифтах, включающих ее, этот символ должен быть помещен в блок Unicode PUA (Private User Area, область частного использования);
- это снижает доступность текста, в том числе для копирования и вставки, поиска и программ голосового чтения экрана. Многие приложения достаточно умны, чтобы правильно обрабатывать такие символы, но далеко не все. Поиск может даже сломаться в некоторых браузерах.

Определенно, в наше время должен существовать способ получше!

На самом деле скромный амперсанд (&), который мы все знаем и любим, происходит от лигатуры букв **E** и **t** (*et* на латыни, что означает «и»).



Рис. 5.16. Дискретные лигатуры, которые можно найти во многих профессиональных шрифтах с засечками

Рис. 5.17. Использование жестко закодированных лигатур часто приводит к ужасным результатам, если в используемом шрифте не предусмотрен глиф для данной лигатуры

Решение

В **CSS Fonts Level 3** (http://w3.org/TR/css3-fonts) старое доброе свойство `font-variant` было **преобразовано в сокращение**, включающее множество новых полных свойств. Одно из них — это `font-variant-ligatures`, разработанное специально с целью включения и выключения лигатур. Для того чтобы включить **все возможные лигатуры**, необходимо использовать **три идентификатора**:

```
font-variant-ligatures: common-ligatures
                        discretionary-ligatures
                        historical-ligatures;
```

Это свойство наследуется. Если вы вдруг обнаружите, что дискретные лигатуры ухудшают читабельность, то можете выключить только их. Для этого используйте:

```
font-variant-ligatures: common-ligatures;
```

Вы можете даже явно выключить оставшиеся два типа:

```
font-variant-ligatures: common-ligatures
                        no-discretionary-ligatures
                        no-historical-ligatures;
```

`font-variant-ligatures` также принимает значение `none`, которое отключает любые типы лигатур. **Не используйте `none`, если только у вас нет абсолютного понимания того, что вы делаете.** Чтобы сбросить `font-variant-ligatures` до первоначального значения, следует использовать `normal`, а не `none`.

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**ligatures**

СВЯЗАННЫЕ СПЕЦИФИКАЦИИ

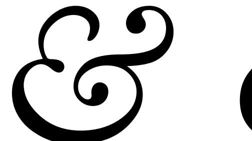
CSS Fonts: http://w3.org/TR/css-fonts

25 Причудливые амперсанды

ПРЕДВАРИТЕЛЬНЫЕ ТРЕБОВАНИЯ

Базовые навыки внедрения шрифтов посредством правил `@font-face`

Проблема



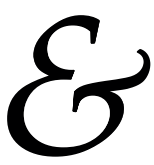



Рис. 5.18. Несколько милых амперсандов в шрифтах, доступных по умолчанию на большинстве компьютеров. Слева направо: Baskerville, Goudy Old Style, Garamond, Palatino (все курсивные)

Вы наверняка заметили, что в книгах, посвященных типографике, скромный амперсанд всегда превозносится до небес. Никакой другой символ не придает тексту столько элегантности, сколько изящный амперсанд. Целые веб-сайты посвящают поиску шрифтов с самыми красивыми амперсандами. Однако найденный шрифт необязательно будет тем, какой вы хотели бы использовать для всего остального текста. В конце концов, действительно красивый и элегантный эффект в заголовках создается **контрастом между симпатичным шрифтом без засечек и прелестными, замысловатыми амперсандами из шрифтов с засечками**.

Веб-дизайнеры осознали это уже довольно давно, но техника достижения этого эффекта остается кустарной и трудоемкой. Чаще всего амперсанд приходится

оборачивать в `<span>`, делая это либо посредством сценария, либо вручную, например:

HTML

```
HTML <span class="amp">&amp;</span> CSS
```

Затем мы задаем желаемые настройки стиля шрифта только к классу `.amp`:

```
.amp {
    font-family: Baskerville, "Goudy Old Style",
                 Garamond, Palatino, serif;
    font-style: italic;
}
```

HTML & CSS

HTML & CSS

Рис. 5.19. Наш заголовок HTML & CSS до и после украшения замысловатым амперсандом

Это прекрасно работает, и пример оформления текста до и после применения данного решения вы можете видеть на рис. 5.19. Однако это грязная техника, а в некоторых ситуациях, когда у нас нет возможности с легкостью редактировать разметку HTML (например, при использовании CMS), ее и вовсе применить невозможно. Нельзя ли просто приказать CSS применять другие стили к определенным символам?

Решение

Оказывается, для стилизации определенных символов (и даже диапазонов символов) мы действительно можем использовать другой шрифт, но способ достижения этого эффекта не так прост, как хотелось бы.

Обычно мы определяем несколько шрифтов (стеки шрифтов) в объявлениях `font-family`, для того чтобы в ситуациях, когда предпочтительный шрифт недоступен, браузер мог использовать другие шрифты, также подходящие нашему дизайну. Однако многие разработчики забывают, что **это также работает и для отдельных символов**. Если шрифт доступен, но содержит лишь несколько символов, то он будет использоваться для отображения только этих символов, а для всех остальных символов браузер будет использовать резервные шрифты. Это верно как **для локальных, так и для внедряемых шрифтов**, добавляемых посредством правил `@font-face`.

Следовательно, если у нас есть шрифт с **одним только символом** (догадайтесь каким!), то он будет использован только для отображения данного символа, а все остальные символы получат второй, третий или последующий шрифт из нашего стека шрифтов. Таким образом, у нас есть простой способ стилизации

амперсандов: нужно создать веб-шрифт, включающий только нужный нам амперсанд, добавить его с помощью `@font-face`, а затем указать первым в стеке шрифтов:

```
@font-face {
    font-family: Ampersand;
    src: url("fonts/ampersand.woff");
}

h1 {
    font-family: Ampersand, Helvetica, sans-serif;
}
```

Хотя это **гибкое** решение, оно субоптимально, если наша цель — использовать для стилизации амперсандов один из **встроенных шрифтов**. Создавать файл шрифта — уже большая морока, кроме того, это добавляет еще один HTTP-запрос, не говоря уже о потенциальных юридических проблемах в случаях, когда требуемый шрифт не допускает подмену символов. **Существует ли способ использовать локальные шрифты?**

Вы, вероятно, знаете, что дескриптор `src` в правилах `@font-face` также принимает функцию `local()`, предназначенную для указания **имен локальных шрифтов**. Следовательно, вместо отдельного веб-шрифта вы могли бы определить стек локальных шрифтов:

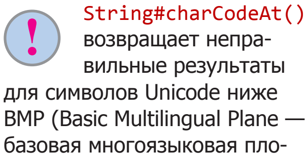
```
@font-face {
    font-family: Ampersand;
    src: local('Baskerville'),
         local('Goudy Old Style'),
         local('Garamond'),
         local('Palatino');
}
```

Однако если вы попытаетесь сейчас применить шрифт Ampersand, то заметите, что теперь **весь текст** выводится с использованием нашего шрифта с засечками (рис. 5.20), так как перечисленные шрифты включают все символы. Это не означает, что мы пошли по неверному пути. Мы просто **забыли дескриптор**, позволяющий объявить, что из этих локальных шрифтов нас интересует только глиф амперсанда. Такой дескриптор существует, он называется `unicode-range`.

HTML &
CSS

Рис. 5.20. Добавление локальных шрифтов посредством `@font-face` приводит к тому, что они по умолчанию используются для оформления всего текста

Дескриптор `unicode-range` работает только внутри правил `@font-face` (отсюда и термин «дескриптор»;



String#charCodeAt() возвращает неправильные результаты для символов Unicode ниже BMP (Basic Multilingual Plane — базовая многоязыковая плоскость). Однако 99,9% символов, которые вам когда-либо потребуются, находятся в этой плоскости. Если вы получаете результат в диапазоне D800-DFFF, это означает, что у вас «астральный» символ и вам лучше прибегнуть к помощи надежного сетевого инструмента для выяснения, какова его кодовая точка Unicode. Метод ES6 String#codePointAt() решает эту проблему.

это **не** свойство CSS) и ограничивает используемые символы определенным поднабором. Он работает как с локальными, так и с удаленными шрифтами. Многие браузеры даже настолько умны, что не загружают удаленные шрифты, если соответствующие символы на странице отсутствуют!

К сожалению, синтаксис unicode-range настолько же **сложен**, насколько полезен сам этот дескриптор. Он работает с *кодовыми точками Unicode*, а не с буквенными символами. Следовательно, прежде чем применять его, необходимо найти шестнадцатеричную кодовую точку символов, которые вы хотели бы указать в дескрипторе. Для этого существует множество сетевых инструментов, или же вы можете просто воспользоваться следующим фрагментом JS-кода в консоли:

JS

```
"&".charCodeAt(0).toString(16); // возвращает 26
```

Теперь, зная шестнадцатеричные кодовые точки, вы можете добавлять к ним спереди U+, задавая таким образом отдельные символы. Вот как наше объявление будет выглядеть для сценария с амперсандом:

```
unicode-range: U+26;
```

Если вам необходимо указать **диапазон** символов, то для этого все так же требуется только один префикс U+, например U+400-4FF. В действительности для такого типа диапазона вы могли бы даже использовать подстановочные символы и записывать его как U+4??. Множественные символы и диапазоны также допустимы, но их нужно разделять запятыми; например: U+26, U+4??, U+2665-2670. Нам же достаточно одного символа. Наш код теперь выглядит так:

```
@font-face {
    font-family: Ampersand;
    src: local('Baskerville'),
         local('Goudy Old Style'),
         local('Palatino'),
         local('Book Antiqua');
    unicode-range: U+26;
}

h1 {
    font-family: Ampersand, Helvetica, sans-serif;
}
```

Протестировав его (рис. 5.21), вы увидите, что нам действительно удалось применить другой шрифт к амперсандам! Однако результат все же еще не совсем тот, которого мы ожидали. Амперсанд на рис. 5.19 был из курсивного варианта шрифта Baskerville, ведь чаще всего именно **в курсивных шрифтах с засечками амперсанды бывают намного симпатичнее**. Но мы не стилизуем непосредственно сами амперсанды, так как же нам использовать курсивное начертание?

HTML & CSS

Рис. 5.21. Применение другого шрифта к нашим амперсандам с помощью стека шрифтов и дескриптора `unicode-range`

Одной из первых идей могло бы стать использование дескриптора `font-style` в правиле `@font-face`. Однако это не даст желаемого результата. Дескриптор всего лишь приказывает браузеру использовать эти шрифты для курсивного текста. Следовательно, наш шрифт Ampersand будет полностью проигнорирован, если только вся строка заголовка не будет оформлена курсивом (и тогда мы действительно увидим симпатичный курсивный амперсанд).

Чтобы узнать PostScript-имя шрифта в Mac OS X, выберите его в приложении FontBook и нажмите ⌘I.

К сожалению, единственным решением здесь остается небольшой трюк: вместо названия семейства шрифтов мы воспользуемся *PostScript-именем* **конкретного стиля/конкретной насыщенности шрифта**, который нам требуется. Таким образом, для получения курсивных версий используемых шрифтов нам нужен такой код:

```
@font-face {
    font-family: Ampersand;
    src: local('Baskerville-Italic'),
         local('GoudyOldStyleT-Italic'),
         local('Palatino-Italic'),
         local('BookAntiqua-Italic');
    unicode-range: U+26;
}

h1 {
font-family: Ampersand, Helvetica, sans-serif;
}
```

И это, наконец, дает нам желаемые амперсанды в точности как на рис. 5.19. К сожалению, если нам потребуется дополнительная настройка стилей для

амперсандов (скажем, мы захотим увеличить кегль шрифта, снизить непрозрачность или сделать что-то еще), то придется пойти по пути использования HTML-элемента. Но если все, что нам нужно, — это другой шрифт и другой стиль/другая насыщенность шрифта, то этот трюк способен на настоящие чудеса! Тот же общий принцип можно использовать для стилизации с использованием других шрифтов чисел, символов и знаков препинания — возможности бесконечны!

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**ampersands**



Благодарности

Спасибо **Дрю Маклиллану** (http://allinthehead.com) за первую версию этого эффекта (http://24ways.org/2011/creating-customfont-stacks-with-unicode-range).

СВЯЗАННЫЕ СПЕЦИФИКАЦИИ

CSS Fonts: http://w3.org/TR/css-fonts

26 Настройки подчеркивания

ПРЕДВАРИТЕЛЬНЫЕ ТРЕБОВАНИЯ

Градиенты CSS, свойство `background-size`, свойство `text-shadow`, секрет «Фон в полоску»

Проблема

Дизайнеры — большие привереды. Мы всегда пытаемся переделать все под себя и очень щепетильны в реализации идей, добиваясь максимального соответствия с видением и стремясь создавать интуитивно понятные и простые в использовании дизайны. **Значения по умолчанию редко бывают достаточно хороши для нас.**

Подчеркивание текста — одна из тех вещей, которые нам очень нравится подгонять под свой вкус. Хотя стандартное подчеркивание также работает неплохо, чаще всего оно смотрится слишком навязчиво, не говоря уже о том, что **в разных браузерах оно визуализируется по-разному**. И несмотря на то что подчеркивание текста доступно нам со времен зарождения Сети, у нас никогда не было возможности настраивать его вид. Даже появление CSS дало нам только обыкновенный выключатель — использовать подчеркивание или нет:

```
text-decoration: underline;
```

Как обычно, когда нам не предоставляют нужных инструментов, мы сами придумываем трюки. У нас не было возможности настраивать вид подчеркивания текста, поэтому мы начали имитировать его с помощью рамок — не исключено, что это вообще самый первый трюк CSS, придуманный дизайнерами:

```
a[href] {
    border-bottom: 1px solid gray;
    text-decoration: none;
}
```

Несмотря на то что, имитируя подчеркивание текста с помощью `border-bottom`, мы можем управлять цветом, толщиной и стилем линии, это решение далеко от идеала. Как видно на рис. 5.22, такие «подчеркивания» находятся слишком далеко от текста — даже под нижними выносными элементами глифов! Можно было бы попытаться решить проблему, определив для ссылок свойство `display` со значением `inline-block` и меньшую величину `line-height`, например так:

“The only way to get rid of a temptation is to yield to it.”

Рис. 5.22. Ложные подчеркивания, созданные с помощью `border-bottom`

```
display: inline-block;
border-bottom: 1px solid gray;
line-height: .9;
```

Это работает — подчеркивание становится ближе к тексту, — но **перенос слов на новую строку при этом работает неправильно**, как демонстрирует рис. 5.23.

Насколько ближе? На толщину линии, ведь единственное отличие этого метода состоит в том, что подчеркивание рисуется **внутри** поля.

Современный дизайнер мог бы попробовать применить для имитации подчеркивания внутреннюю тень, определяемую посредством `box-shadow`:

```
box-shadow: 0 -1px gray inset;
```

“The only way to get rid of a tempta-tion is to yield to it.”

Рис. 5.23. Попытка справиться с проблемой подчеркиваний, создаваемых нижним краем рамки, работает, но только до тех пор, пока не возникает необходимость перенести текст на новую строку, — и тогда начинается полнейшая неразбериха

Однако при этом возникают те же сложности, что и с `border-bottom`, за исключением того, что подчеркивание отображается чуть ближе к тексту. Существует ли другой способ получать правильные, гибкие и поддающиеся тонкой настройке подчеркивания?

Решение

Часто лучшие решения можно обнаружить в самых неожиданных местах. В данном случае оно пришло в форме `background-image` и связанных свойств. Возможно, вы думаете, что это какое-то безумие, но проявите чуточку терпения. Фоны идеально обтекают текст, даже когда он переносится на новую строку, а благодаря новым свойствам, которые мы получили в **CSS Backgrounds & Borders Level 3**, таким как `background-size`, мы можем с высокой точностью контролировать их вид и поведение. Нам даже не требуются отдельные HTTP-запросы для загрузки

“The only way to get rid of a temptation is to yield to it.”

Рис. 5.24. Наши аккуратно сработанные уникальные подчеркивания, созданные с помощью градиентов CSS

фоновых изображений, так как мы можем генерировать их на лету с помощью градиентов CSS:

```
background: linear-gradient(gray, gray)
            no-repeat;
background-size: 100% 1px;
background-position: 0 1.15em;
```

“The only way to get rid of a temptation is to yield to it.”

Рис. 5.25. Наши уникальные подчеркивания, которым `text-shadow` не дает пересекать нижние выносные элементы букв

Каким **элегантным** и **ненавязчивым** получается результат, можно видеть на рис. 5.24. Однако и здесь есть еще потенциал для небольшого улучшения. Обратите внимание, как подчеркивание **пересекает нижние выносные элементы** букв p и y. Не правда ли, было бы намного лучше, если бы вокруг них было немного пустого пространства? Если наш фон залит сплошным цветом, то имитировать пустое пространство можно посредством двух теней `text-shadow`, также использующих сплошной цвет, совпадающий с цветом фона (рис. 5.25):

```
background: linear-gradient(gray, gray) no-repeat;
background-size: 100% 1px;
background-position: 0 1.15em;
text-shadow: .05em 0 white, -.05em 0 white;
```

БУДУЩЕЕ.
ПОДЧЕРКИВАНИЕ ТЕКСТА В БУДУЩЕМ

В будущем для настройки внешнего вида наших подчеркиваний нам не придется полагаться на подобные трюки. В CSS Text Decoration Level 3 (http://w3.org/TR/css-text-decor-3) запланировано несколько свойств специально для решения этой задачи, в частности:

- `text-decoration-color` для настройки цвета подчеркиваний и других элементов их художественного оформления;
- `text-decoration-style` для настройки стиля оформления (например, сплошная линия, пунктирная, волнистая и т. п.);
- `text-decoration-skip` для того, чтобы пропускать пробелы, нижние выносные элементы букв и другие объекты;
- `text-underline-position` для тонкой настройки точного местоположения подчеркивания.

Однако в настоящее время эти свойства практически не поддерживаются браузерами.

“The only way to get rid of a temptation is to yield to it.”

Рис. 5.26. Полностью настроенные в соответствии с нашим вкусом пунктирные подчеркивания, для создания которых использовались градиенты CSS

В решении, основанном на градиентах, лучше всего то, что градиенты **невероятно гибкие**. Например, вы могли бы определить пунктирное подчеркивание (рис. 5.26) с помощью такого кода:

```
background: linear-gradient(90deg,
            gray 66%, transparent 0) repeat-x;
background-size: .2em 2px;
background-position: 0 1em;
```

Соотношение между величиной штрихов и промежутков можно контролировать с помощью позиций границ перехода цвета, а размер этих составляющих — с помощью свойства `background-size`.

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**underlines**

В качестве упражнения вы можете попробовать создать **красное волнистое подчеркивание**, как то, которое используется для **подсветки грамматических ошибок** (*подсказка: вам потребуются два градиента*). Решение вы найдете в следующем примере из врезки «Попробуйте сами!», но, пожалуйста, постарайтесь не подглядывать, не попробовав решить задачу самостоятельно, — так гораздо интереснее и увлекательнее!

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**wavy-underlines**



Спасибо Марсин Вичари (http://aresLuna.org) за первую версию этого эффекта (http://medium.com/designing-medium/craftinglink-underlines-on-medium-7c03a9274f9).

СВЯЗАННЫЕ СПЕЦИФИКАЦИИ

CSS Backgrounds & Borders: http://w3.org/TR/css-backgrounds

CSS Image Values: http://w3.org/TR/css-images

CSS Text Decoration: http://w3.org/TR/css-text-decor

27 Реалистичные текстовые эффекты

ПРЕДВАРИТЕЛЬНЫЕ ТРЕБОВАНИЯ

Базовые навыки использования `text-shadow`

Проблема

Порой определенные украшательства текста получают в Сети огромное распространение. Например, вдавленный текст, размытие текста при наведении указателя мыши, объемный текст и т. д. Эти эффекты обычно строятся на комбинации тщательно проработанных текстовых, а также на знании того, как устроено наше зрение, — многие из них в той или иной степени основываются

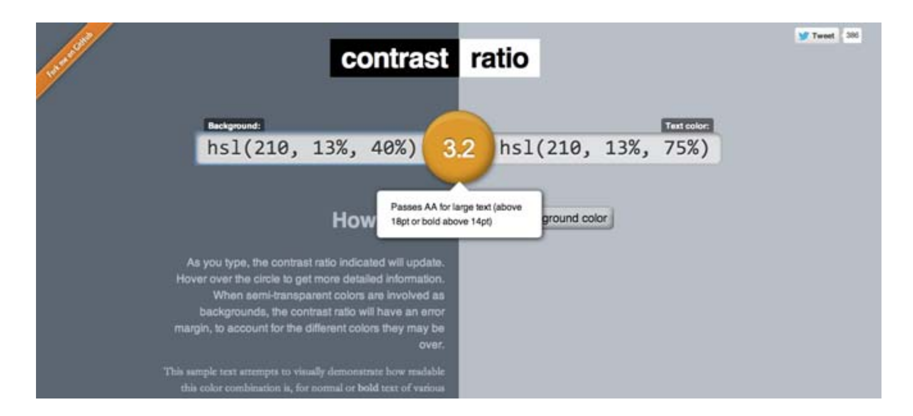


Рис. 5.27. При использовании подобных эффектов очень легко позабыть о доступности текста, так что никогда не ленитесь тестировать степень контрастности оформления (удобный инструмент для этого вы найдете на странице http://leaverou.github.io/contrast-ratio; он принимает все поддерживаемые CSS цветовые форматы)

на **оптических иллюзиях**. Если вам уже известны используемые трюки, то создавать такие эффекты очень легко, однако они не всегда поддаются простому обратному декодированию посредством инструментов разработки.

Секрет посвящен созданию подобных эффектов, для того чтобы вы никогда больше не задавались вопросом: «Как вообще это работает?!»

Эффект вдавленного текста

"The only way to get rid of a temptation is to yield to it."

"The only way to get rid of a temptation is to yield to it."

Рис. 5.28. Эффект вдавленного текста, реализованный для темного шрифта на более светлом фоне (*вверху*: до, *внизу*: после)

Эффект вдавленного текста — один из самых популярных на веб-сайтах со скевоморфным дизайном. И хотя скевоморфный дизайн уже не так популярен, как когда-то, у него всегда будут свои верные поклонники.

Этот эффект лучше всего работает на умеренно светлом фоне с темным текстом, но его можно применять и к светлому тексту на темных фонах, если только текст не на 100% черный, а фон не на 100% белый или черный.

В его основе тот же замысел, который используется со времен первых графических интерфейсов пользователя для создания впечатления вдавленных или выпуклых кнопок: более светлая тень внизу (или темная наверху) создает **иллюзию, что объект выгравирован** на основной поверхности. Схожим образом более темная тень внизу (или светлая наверху) создает **иллюзию того, что объект выдавлен** из основной поверхности. Причина, почему это работает, кроется в том, что мы обычно предполагаем **наличие источника света наверху**: таким образом, выпуклый объект должен отбрасывать тень вниз, а выгравированный объект должен быть освещен снизу.

В качестве точки отсчета давайте возьмем цвета с рис. 5.28. Цвет текста здесь — `hsl(210, 13%, 30%)`, а цвет фона — `hsl(210, 13%, 60%)`:

```
background: hsl(210, 13%, 60%);
color: hsl(210, 13%, 30%);
```

Когда мы используем темный шрифт на более светлом фоне (как в предыдущем примере), **наилучшим образом обычно работает светлая тень внизу**. Насколько светлая — зависит от конкретных цветов, работающих в вашем дизайне, а также от того, насколько заметным должен получиться эффект. Поэкспериментируйте с параметром альфа-канала, чтобы добиться наиболее привлекательного

результата. В нашем примере мы остановились на 80% белом, но в вашем решении значения могут быть совершенно иными:

```
background: hsl(210, 13%, 60%);
color: hsl(210, 13%, 30%);
text-shadow: 0 1px 1px hsla(0,0%,100%,.8);
```

Результат вы можете видеть на рис. 5.28. В данном случае для создания эффекта мы использовали значения в пикселах, а не в единицах `em`, однако если в вашем дизайне текст может быть любого размера, от крошечных до огромных букв, то вам лучше подойдут единицы длины `em`:

```
text-shadow: 0 .03em .03em hsla(0,0%,100%,.8);
```

Что произойдет, если у нас будет светлый текст на темном фоне? Тень, определяемая в фрагменте кода выше, приводит к ужасным результатам в случае, когда цвета меняются местами, из-за чего текст выглядит расплывшимся (рис. 5.29). Означает ли это, что эффект вдавленного текста в данном случае применить невозможно? Нет, это всего лишь означает, что необходимо немного скорректировать подход. В подобных ситуациях темная тень наверху работает лучше, что подтверждает рис. 5.30. CSS-код выглядит так:

```
background: hsl(210, 13%, 40%);
color: hsl(210, 13%, 75%);
text-shadow: 0 -1px 1px black;
```

“The only way to get rid of a temptation is to yield to it.”

Рис. 5.29. Эффект вдавленного текста сломался: мы попробовали применить предыдущее решение к тексту, цвет которого светлее цвета фона

“The only way to get rid of a temptation is to yield to it.”

“The only way to get rid of a temptation is to yield to it.”

Рис. 5.30. Эффект вдавленного текста при использовании светлого цвета шрифта на темном фоне (*вверху*: до, *внизу*: после)

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**letterpress**

Текст с обводкой

В будущем создавать текст с контуром/обводкой будет намного проще, так как мы сможем использовать параметр размазывания свойства `text-shadow`, делая тени крупнее и превращая их в подобие обводки — аналогично тому, как мы



Рис. 5.31. Текст с настоящей обводкой, созданной с помощью параметра размазывания свойства text-shadow

имитируем контуры посредством размазывания box-shadow. К сожалению, поддержка браузерами этого параметра в настоящее время очень ограниченна, и нам приходится полагаться на другие способы имитировать обводку, дающие более или менее приемлемые результаты.

Самый распространенный способ — накладывать друг на друга несколько теней text-shadow с немного отличающимися значениями смещения, например так (результат см. на рис. 5.32):

```
background: deeppink;
color: white;
text-shadow: 1px 1px black, -1px -1px black,
             1px -1px black, -1px 1px black;
```



Рис. 5.32. Ложный контур толщиной 1px, созданный путем наложения друг на друга нескольких теней text-shadow

В качестве альтернативы можно было бы использовать несколько слегка размытых теней без смещения:

```
text-shadow: 0 0 1px black, 0 0 1px black,
             0 0 1px black, 0 0 1px black,
             0 0 1px black, 0 0 1px black;
```

Однако это не всегда позволяет получить визуально привлекательные результаты, к тому же это более дорогостоящий с точки зрения производительности способ, поскольку размытие больше нагружает системные ресурсы.



Рис. 5.33. Ужасная обводка толщиной 3px, созданная с помощью нескольких теней text-shadow с разными значениями смещения

К сожалению, чем толще обводка, тем хуже получаются результаты каждого из этих решений. Например, взгляните, как некрасиво смотрится обводка толщиной 3px (рис. 5.33):

```
background: deeppink;
color: white;
text-shadow: 3px 3px black, -3px -3px black,
             3px -3px black, -3px 3px black;
```

Конечно, для решения задачи всегда можно прибегнуть к помощи формата SVG, но он сильно замусоривает разметку. Предположим, что мы хотим

использовать код SVG для стилизации заголовка первого уровня. HTML-код будет выглядеть так:

SVG

```
<h1><svg width="2em" height="1.2em">
    <use xlink:href="#css" />
    <text id="css" y="1em">CSS</text>
</svg></h1>
```

А CSS-код придется написать примерно такой:

```
h1 {
    font: 500%/1 Rockwell, serif;
    background: deeppink;
    color: white;
}

h1 text {
    fill: currentColor;
}

h1 svg { overflow: visible }

h1 use {
    stroke: black;
    stroke-width: 6;
    stroke-linejoin: round;
}
```

Точно не идеальный вариант, но он дает наилучшие визуальные результаты (рис. 5.34), и даже в древних браузерах, которые не поддерживают SVG, текст все так же остается читабельным, красиво стилизован и распознается программами чтения экрана.

Рис. 5.34. Использование SVG для создания красивой толстой обводки

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**stroked-text**

Сияющий текст

Сияние — довольно распространенный эффект. Часто так отображаются ссылки при наведении на них указателя мыши или же заголовки на определенных



Рис. 5.35. Сияющий текст, созданный с помощью двух простых теней `text-shadow`

веб-сайтах. Кроме того, это один из самых легких в создании эффектов. В своей простейшей форме он требует всего лишь парочки наложенных друг на друга теней `text-shadow` безо всяких сдвигов и того же цвета, что и основной текст (рис. 5.35):

```
background: #203;
color: #ffc;
text-shadow: 0 0 .1em, 0 0 .3em;
```

Если вы создаете эффект для состояния ссылки, когда на нее наводится указатель мыши, то нужно также добавить переход, например так:

```
a {
    background: #203;
    color: white;
    transition: 1s;
}
a:hover {
    text-shadow: 0 0 .1em, 0 0 .3em;
}
```

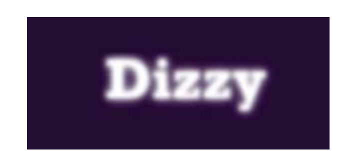


Рис. 5.36. Псевдоразмытый текст; такой эффект получается благодаря тому, что сам текст скрывается, а отображаются только его тени

Вы можете определить еще более интересный эффект, скрывая сам текст при срабатывании `:hover` и, по сути, создавая иллюзию того, что буквы плавно расплываются (рис. 5.36):

```
a {
    background: #203;
    color: white;
    transition: 1s;
}
a:hover {
    color: transparent;
    text-shadow: 0 0 .1em white, 0 0 .3em white;
}
```

Однако помните, что зависимость от `text-shadow` в вопросе отображения текста таит в себе опасность: у этого решения нет элегантного обходного пути. Если `text-shadow` не поддерживается, то вообще никакой текст визуализирован не будет. Таким образом, необходимо проявлять осторожность и применять это только в тех окружениях, которые поддерживают `text-shadow`. Или же размывать текст с помощью фильтров CSS:

```
a {
    background: #203;
    color: white;
    transition: 1s;
}
a:hover {
    filter: blur(.1em);
}
```

Возможно, в этом варианте поддержка браузерами будет хуже, но, по крайней мере, ничего не сломается, если поддержка будет отсутствовать вовсе.

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**glow**

Объемный текст

Еще один популярный (пожалуй, даже слишком) эффект скевоморфного дизайна — это объемный текст (рис. 5.37). Главная идея заключается в использовании большого количества наложенных друг на друга теней: каждая чуть темнее предыдущей, без размытия и со сдвигом только на `1px`. В самом же низу «стопки» должна находиться сильно размытая темная тень, имитирующая тень, которую отбрасывала бы вся эта конструкция.



Рис. 5.37. Объемный текст, созданный посредством наложения друг на друга нескольких теней `text-shadow`

Давайте возьмем в качестве отправной точки текст на рис. 5.38, для стилизации которого применяется этот простой CSS-код:

```
background: #58a;
color: white;
```



Рис. 5.38. Наша отправная точка

Теперь добавим несколько теней `text-shadow`, постепенно делая их темнее:

```
background: #58a;
color: white;
text-shadow: 0 1px hsl(0,0%,85%),
             0 2px hsl(0,0%,80%),
             0 3px hsl(0,0%,75%),
             0 4px hsl(0,0%,70%),
             0 5px hsl(0,0%,65%);
```



Рис. 5.39. Почти готово, но еще не совсем реалистично



Рис. 5.40. Оформление в ретростиле

Как видно на рис. 5.39, мы уже приближаемся к желаемому результату, но эффект все еще смотрится недостаточно реалистично. Верите или нет, но для того, чтобы достичь результата, показанного на рис. 5.37, нам нужно всего лишь добавить еще одну тень внизу:

```
background: #58a;
color: white;
text-shadow: 0 1px hsl(0,0%,85%),
             0 2px hsl(0,0%,80%),
             0 3px hsl(0,0%,75%),
             0 4px hsl(0,0%,70%),
             0 5px hsl(0,0%,65%),
             0 5px 10px black;
```

Такой повторяющийся громоздкий код — первый кандидат на преобразование в препроцессорную примесь. Один из вариантов, как это можно было бы сделать в SCSS, показан далее:

SCSS

```
@mixin text-3d($color: white, $depth: 5) {
    $shadows: ();
    $shadow-color: $color;

    @for $i from 1 through $depth {
        $shadow-color: darken($shadow-color, 10%);
        $shadows: append($shadows,
                  0 ($i * 1px) $shadow-color, comma);
    }

    color: $color;
    text-shadow: append($shadows,
                  0 ($depth * 1px) 10px black, comma);
}

h1 { @include text-3d(#eee, 4); }
```

Существует множество вариаций этого эффекта. Например, если сделать все тени черными (цвет `black`) и убрать последнюю размытую тень, то можно имитировать эффект вдавленного текста, часто используемый для имитации старых табличек или оформления их в ретростиле (рис. 5.40):

```
color: white;
background: hsl(0,50%,45%);
text-shadow: 1px 1px black, 2px 2px black,
             3px 3px black, 4px 4px black,
             5px 5px black, 6px 6px black,
             7px 7px black, 8px 8px black;
```

Этот вариант проще преобразовать в примесь или — что в данном случае более удобно — в функцию:

SCSS

```
@function text-retro($color: black, $depth: 8) {
    $shadows: (1px 1px $color,);
    @for $i from 2 through $depth {
        $shadows: append($shadows,
                  ($i*1px) ($i*1px) $color, comma);
    }

    @return $shadows;
}

h1 {
    color: white;
    background: hsl(0,50%,45%);
    text-shadow: text-retro();
}
```

СВЯЗАННЫЕ СПЕЦИФИКАЦИИ

CSS Text Decoration: http://w3.org/TR/**css-text-decor**

28 Текст по кругу

ПРЕДВАРИТЕЛЬНЫЕ ТРЕБОВАНИЯ

Базовые знания формата SVG

Проблема

Хотя это не самый распространенный эффект, иногда возникает необходимость вывести короткую строку текста вдоль дуги окружности. Но, сталкиваясь с таким требованием, мы осознаем, что помощи от CSS ждать не приходится. Не существует свойства или функции CSS, позволяющей изгибать строки текста таким способом, а те основанные на CSS решения, до которых нам удается додуматься, выглядят настолько грязными, что неловко даже упоминать их. Существует ли способ воплотить этот эффект, не прибегая к помощи изображений и не теряя душевного равновесия и самоуважения?

Решение

Есть несколько сценариев, помогающих решить эту задачу. Принцип их работы заключается в том, что каждая буква оборачивается в отдельный элемент `<span>` и эти элементы по отдельности поворачиваются, что позволяет сформировать окружность. Но это не только ужасно грязный трюк, он вдобавок непомерно раздувает код и добавляет на страницу десятки DOM-элементов, не имеющих практической ценности.



Рис. 5.41. Текст выводится вдоль дуги окружности на кнопках, напоминающих пуговицы, на веб-сайте http://juliancheal.co.uk. Обратите внимание, что это единственный способ добавить текст на кнопки, не сломав метафору пуговиц, ведь центр кнопки занят отверстиями и нитками

Идеального способа достичь желаемого результата с помощью чистого CSS не существует, но мы можем с легкостью создавать подобные дизайны, используя **немного строкового SVG**. Формат SVG всегда поддерживал отображение текста вдоль любого пути, а дуги окружности — это всего лишь один из вариантов формы пути. Почему бы не попробовать?

К сожалению, `<textPath>` работает только с элементами `<path>`, поэтому для создания нашей окружности у нас не получится использовать гораздо более понятный элемент `<circle>`.

Простейший способ нарисовать текст по кругу с помощью SVG — поместить его в элемент `<textPath>` внутри элемента `<text>`. Элемент `<textPath>` также ссылается на элемент `<path>`, определяя форму пути по его идентификатору. Текст внутри строкового SVG также наследует большую часть стилизации шрифтов (за исключением `line-height`, так как это в SVG задается вручную), поэтому, в отличие от решений, включающих внешние изображения в формате SVG, в данном случае о стилях можно не беспокоиться.

Предположим, мы хотим вывести фразу *circular reasoning works because* по кругу, чтобы она формировала замкнутую окружность, как на рис. 5.42. Начнем с добавления строкового SVG внутри нашего элемента HTML, а также с определения пути, описывающего окружность:



Рис. 5.42. Конечный результат, которого мы хотим добиться

SVG

```
<div class="circular">
    <svg viewBox="0 0 100 100">
        <path d="M 0,50 a 50,50 0 1,1 0,1 z"
              id="circle" />
    </svg>
</div>
```

Обратите внимание, что мы определили габариты посредством `viewBox`, а не с помощью атрибутов `width` и `height`. Это позволяет настраивать систему координат и соотношение сторон рисунка, вместо того чтобы всегда использовать четко определенный размер. Так не только намного компактнее; это экономит несколько строк CSS-кода, поскольку нам больше не приходится определять равную 100% ширину и высоту для элемента `<svg>` — он сам подстраивается под размер своего контейнера.

Если вы не понимаете синтаксис пути, не беспокойтесь. **Его вообще мало кто понимает**, и даже те, кто был посвящен в таинство синтаксиса пути в SVG, чаще всего забывают о нем в течение нескольких минут. Если вам интересно, то вот три команды, которые включает этот невероятно загадочный синтаксис:

- `M 0,50`: перейти в точку (0,50);
- `a 50,50 0 1,1 0,1`: нарисовать дугу из точки, в которой вы находитесь в данный момент, в точку, которая находится на `0` единиц правее и на `1` единицу ниже вашей текущей позиции. Радиус этой дуги равен `50`, как по горизонтали, так и по вертикали. Из двух возможных углов выбрать наибольший и из двух возможных дуг выбрать ту, что находится справа от двух точек, а не слева;
- `z`: закрыть путь прямым отрезком.

Пока что наш путь представляет собой всего лишь черную окружность (рис. 5.43). Мы добавляем текст с помощью элементов `<text>` и `<textPath>` и связываем его с нашей окружностью посредством свойства `xlink:href`, как в следующем фрагменте кода:

SVG

```
<div class="circular">
    <svg viewBox="0 0 100 100">
        <path d="M 0,50 a 50,50 0 1,1 0,1 z"
              id="circle" />
        <text><textPath xlink:href="#circle">
            circular reasoning works because
        </textPath></text>
    </svg>
</div>
```

Как видно на рис. 5.44, хотя нам предстоит еще немало потрудиться, чтобы сделать этот текст привлекательным и читабельным, мы уже достигли результата, которого с помощью чистого CSS не сумели бы добиться и за миллион лет!

Следующим шагом будет **удаление черной заливки из нашего кругового пути**. Мы вообще не хотим, чтобы какая-либо часть нашей окружности была видна; ее единственное предназначение — служить направляющей для нашего текста. Этого можно добиться несколькими разными способами: например, поместив наш контур в элемент `<defs>` (придуманный как раз для этой цели). Однако при создании нашего эффекта мы хотим минимизировать объем SVG-разметки, поэтому применим решение из CSS, а именно `fill: none`:

```
.circular path { fill: none; }
```

Теперь, когда черный круг исчез, мы можем внимательнее изучить остальные недостатки. Самая большая проблема заключается в том, что **большая часть текста находится за пределами SVG-элемента** и **обрезается** им. Чтобы исправить этот дефект, нужно сделать контейнер меньше и применить к SVG-элементу `overflow: visible`, чтобы он не обрезал никакое содержимое за пределами своего окна просмотра:

```
.circular {
    width: 30em;
    height: 30em;
}

.circular svg {
    display: block;
    overflow: visible;
}
```

Результат вы видите на рис. 5.46. Это почти то, что нам нужно, но часть текста все равно обрезается. Причина в том, что на обтекание влияют только габаритные размеры SVG-элемента, но не то, насколько

Почему синтаксис пути в SVG такой запутанный? В те времена, когда он разрабатывался, люди были уверены, что никто не будет писать SVG-код вручную, поэтому рабочая группа SVG стремилась к максимально компактному синтаксису, уменьшая размер файла.

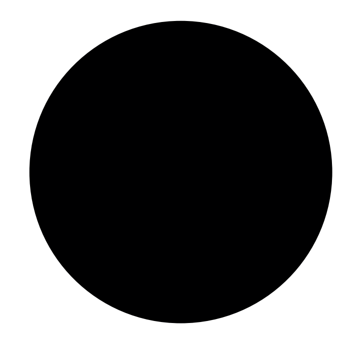


Рис. 5.43. Наш путь сейчас выглядит как окружность с цветом заливки по умолчанию (`black`)



Рис. 5.44. Хотя предстоит еще немало работы, мы уже совершили то, на что чистый CSS просто не способен

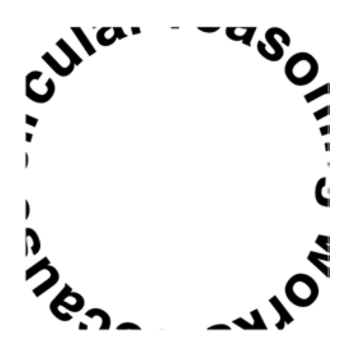


Рис. 5.45. После того как мы сделали путь невидимым, остальные проблемы стали заметнее

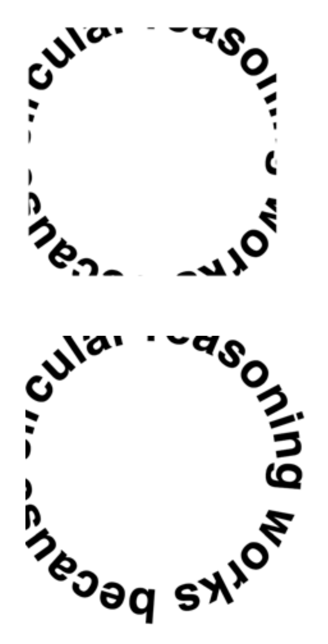


Рис. 5.46. *Вверху*: определение ширины и высоты нашего элемента-контейнера. *Внизу*: добавление `overflow: visible`

содержимое выходит за пределы окна просмотра. Следовательно, тот факт, что какой-то текст переливается через края контейнера, создаваемого элементом `<svg>`, не заставляет этот SVG-элемент сдвинуться вниз. Нам придется сделать это вручную, с помощью поля:

```
.circular {
    width: 30em;
    height: 30em;
    margin: 3em auto 0;
}

.circular svg {
    display: block;
    overflow: visible;
}
```

Вот и все! Результат выглядит в точности как на рис. 5.42, и распознавание текста программами чтения экрана проходит без сложностей. Если у нас только один фрагмент текста, нарисованного вдоль дуги окружности (скажем, на логотипе веб-сайта), то работа закончена. Но если нужно применить подобную стилизацию к нескольким элементам на странице, то хотелось бы **избежать повторения всей этой разметки SVG**. Для чего можно написать короткий сценарий, который будет автоматически генерировать необходимые SVG-элементы, встречая в разметке нечто подобное:

HTML

```
<div class="circular">
    circular reasoning works because
</div>
```

Наш код будет проходить по всем элементам с классом `circular`, удаляя их текст и сохраняя его в переменной, а также добавляя необходимые SVG-элементы:

JS

```
$$('.circular').forEach(function(el) {
    var NS = "http://www.w3.org/2000/svg";
    var xlinkNS = "http://www.w3.org/1999/xlink";
    var svg = document.createElementNS(NS, "svg");
    var circle = document.createElementNS(NS, "path");
    var text = document.createElementNS(NS, "text");
    var textPath = document.createElementNS(NS, "textPath");
    svg.setAttribute("viewBox", "0 0 100 100");

    circle.setAttribute("d", "M0,50 a50,50 0 1,1 0,1z");
    circle.setAttribute("id", "circle");

    textPath.textContent = el.textContent;
    textPath.setAttributeNS(xlinkNS, "xlink:href", "#circle");

    text.appendChild(textPath);
    svg.appendChild(circle);
    svg.appendChild(text);
    el.textContent = '';
    el.appendChild(svg);
});
```

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**circular-text**

СВЯЗАННЫЕ СПЕЦИФИКАЦИИ

Scalable Vector Graphics (SVG): http://w3.org/TR/SVG

6

Взаимодействие с пользователем

29 Выбор правильного указателя мыши

Проблема

Назначение указателя мыши — не просто показывать, в какой точке экрана находится курсор, но также сообщать пользователю, какие действия ему доступны. Об этой распространенной практике проектирования пользовательских интерфейсов настольных приложений в веб-приложениях часто забывают.

Но вина лежит не только на разработчиках. Во времена CSS 2.1 у нас попросту не было доступа ко многим встроенным курсорам. В основном мы использовали свойство `cursor` для указания, что на чем-то можно щелкнуть, дополняя его курсором `pointer`, или же иногда добавляли всплывающие подсказки с помощью курсора `help`. Некоторые также применили курсоры `wait` и `progress` вместо или в дополнение к указателю загрузки. Но этим дело и ограничивалось. И хотя в **CSS User Interface Level 3** (http://w3.org/TR/css3-ui/#cursor) мы получили целую пачку новых встроенных курсоров, большинство разработчиков продолжают в этом вопросе придерживаться старых привычек. Как это часто бывает при работе над взаимодействием с пользователем, вы в действительности не осознаете существования проблемы до тех пор, пока не сталкиваетесь с решением. Позвольте мне показать вам эти решения!

Решение

Полный список новых встроенных курсоров представлен на рис. 6.2, а прочитать об их предназначении вы можете в спецификации. Понятно, что необходимость в таких курсорах существует далеко не во всех веб-приложениях. Например,

в набор входит даже курсор `cell`, «*указывающий, что ячейка или набор ячеек могут быть выделены*». Сложно вообразить себе применение подобного курсора за пределами электронных таблиц и редактируемых сеток.

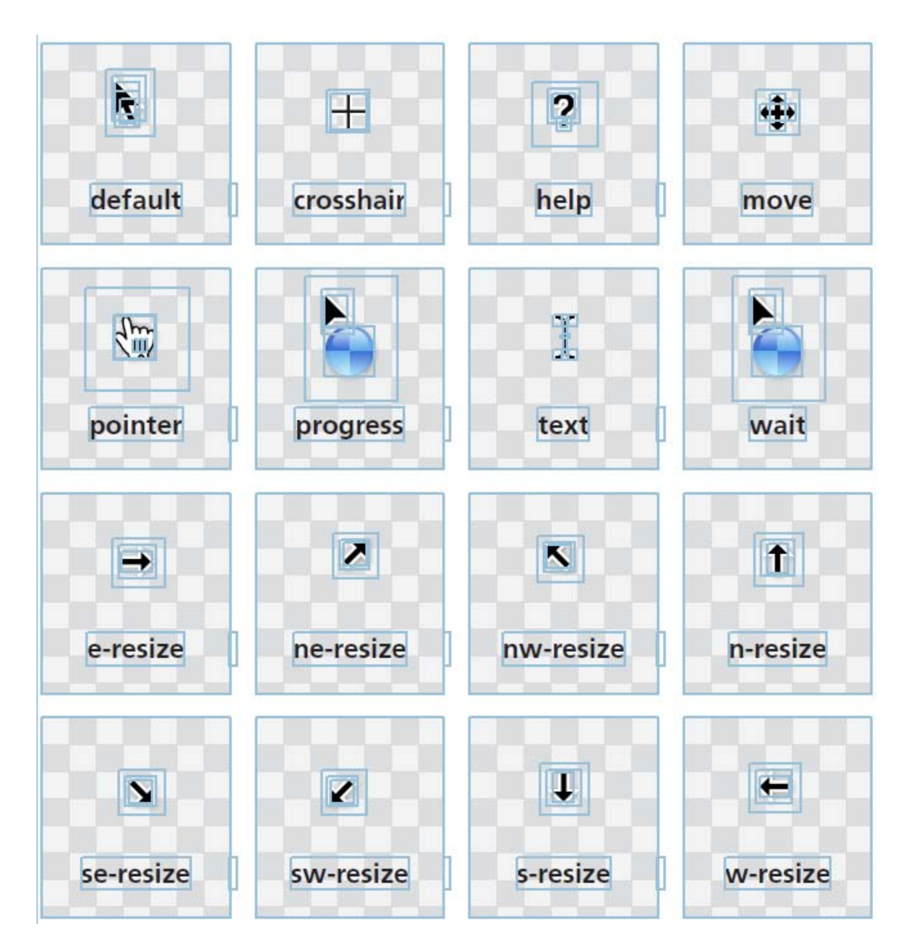


Рис. 6.1. Набор встроенных курсоров в CSS 2.1 был довольно ограниченным (курсоры показаны в том виде, как они отображаются в OS X)

Я не ставлю себе целью посредством этого секрета предложить вам исчерпывающее руководство по возможным вариантам использования всех этих новых курсоров. Однако некоторые из них действительно весьма примечательны и способны моментально повысить удобство использования большого числа веб-приложений, а вам для этого потребуется добавить совсем немного кода.

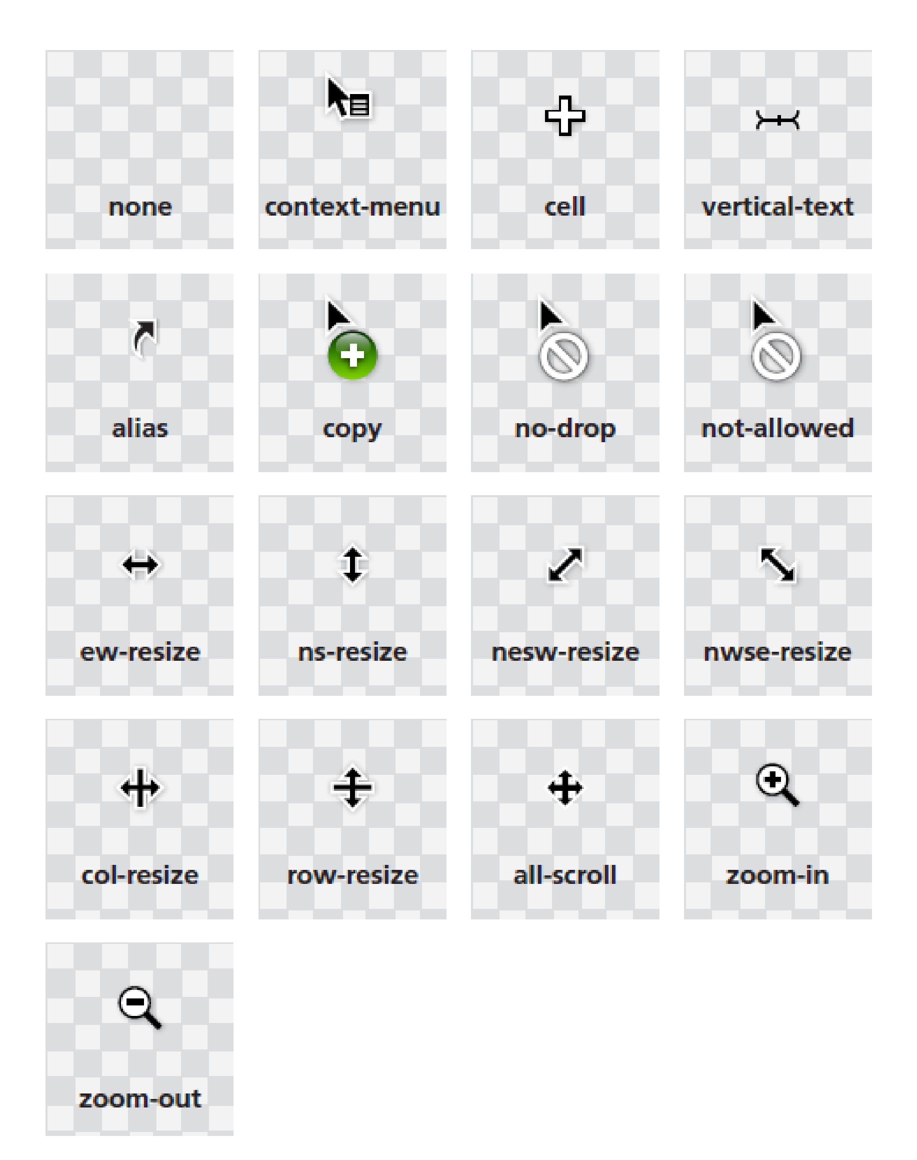


Рис. 6.2. Новые встроенные курсоры, которые мы получили в рамках CSS User Interface Level 3 (http:// w3.org/TR/css3-ui/#cursor) (курсоры показаны в том виде, как они отображаются в OS X)

Обозначение нерабочего состояния

Возможно, наиболее полезным среди новинок является курсор `not-allowed` (рис. 6.3). Он чрезвычайно полезен для указания того, что взаимодействие с определенным элементом управления невозможно по той или иной причине — чаще всего потому, что элемент управления отключен. Сегодня, когда

большинство форм на веб-страницах подвергаются масштабной стилизации, зачастую бывает трудно отличить доступный элемент управления от недоступного, и подобный курсор становится просто незаменимым помощником. Никакой особой настройки он не требует:

```
:disabled, [disabled], [aria-disabled="true"] {
    cursor: not-allowed;
}
```

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**disabled**

Скрывание курсора

Невидимый курсор звучит как ночной кошмар инженера по удобству использования. Как кому-то вообще может прийти в голову такое и почему веб-стандарты упрощают для подобных людей эту задачу? Прежде чем вы начнете поносить людей, очевидно, испытывающих личную субъективную неприязнь к удобству в использовании, вспомните, как вы пытались пользоваться этими ужасными общественными сенсорными экранами (например, в информационных будках или на дисплеях, встроенных в сиденья самолета), а разработчики забывали спрятать указатель мыши, и этот несчастный курсор болтался по экрану, замирая в самых неудобных местах. Или как вам приходилось перемещать мышь к правому краю экрана во время просмотра видео, потому что курсор настырно висел прямо поверх картинки.



Рис. 6.3. Использование курсора not-allowed в качестве подсказки, что элемент управления отключен



Если вы скрываете курсор, когда он находится поверх видео, убедитесь, что не скроете его случайно, когда он окажется над элементами управления. В противном случае этим усовершенствованием вы причините больше вреда, чем принесете пользы.

Очевидно, существует множество сценариев использования, в которых **скрывание курсора способно улучшить впечатление пользователя от взаимодействия с интерфейсом**. Вот почему среди новых ключевых слов для курсора появилось `none`. Скрывать курсор можно было и в CSS 2.1, но для этого требовалось прозрачное изображение в формате GIF размером 1×1, вот так:

```
video {
    cursor: url(transparent.gif);
}
```

Сегодня нам подобные хитрости не требуются, так как мы можем использовать простое `cursor: none`. Однако обеспечить обходной путь часто бывает полезно, так как не все браузеры еще поддерживают курсоры Level 3. Это легко сделать с помощью каскадных стилей:

```
cursor: url('transparent.gif');
cursor: none;
```

СВЯЗАННЫЕ СПЕЦИФИКАЦИИ

CSS Basic User Interface: http://w3.org/TR/css3-ui

30 Расширение области, реагирующей на щелчок мыши

Проблема

Если вы интересуетесь вопросами взаимодействия с пользователем, то наверняка слышали о *законе Фиттса*. Впервые предложенный американским психологом Полом Фиттсом (Paul Fitts) еще в 1954 году, закон Фиттса гласит, что **время, необходимое для быстрого перемещения в целевую область, представляет собой логарифмическую функцию отношения между расстоянием до цели и шириной цели**. Его наиболее часто используемая математическая формулировка выражается как $T = a + b\log_2\left(1+\frac{D}{W}\right)$, где T — затраченное время, D — расстояние до центра цели, W — ширина цели, а a и b — константы.

СОВЕТ

Пронаблюдать закон Фиттса в действии посредством интерактивной визуализации вы можете на странице http://simonwallner.at/ext/fitts.

Хотя графические интерфейсы пользователя в то время еще не существовали, закон Фиттса распространяется в том числе и на координатно-указательные устройства, а сегодня стал самым известным принципом человеко-машинного взаимодействия (Human-Computer Interaction, HCI). Возможно, поначалу это может казаться чем-то удивительным, но помните, что закон Фиттса в большей степени относится к человеческим моторным навыкам, чем к конкретному аппаратному обеспечению.

Очевидное следствие данного закона состоит в том, что чем больше цель, тем проще в нее попасть. Следовательно, **расширение области, реагирующей на щелчки мыши (области попадания)**, вокруг небольших элементов управления, с которыми в противном случае может быть сложно взаимодействовать (а увеличить их невозможно), зачастую **повышает удобство использования**.

Это становится тем важнее, чем большую популярность завоевывают сенсорные экраны. **Никому не нравится тыкать пальцем в экран десять раз подряд, чтобы попасть в эту мерзкую маленькую кнопочку**, и все же подобные ситуации возникают каждый день.

В других сценариях мы хотим, чтобы элемент выдвигался, когда пользователь подводит указатель мыши к краю окна, — как, например, в случае с автоматически скрываемым заголовком, который выскальзывает из-под верхней кромки при приближении указателя мыши. Это также включает необходимость увеличения области попадания (но только в одном направлении). Возможно ли это средствами чистого CSS?

Решение

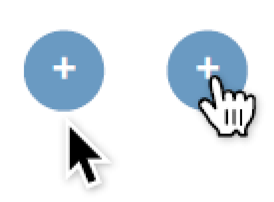


Рис. 6.4. Наша отправная точка с двумя состояниями: когда курсор находится на кнопке (*справа*) и когда он находится ниже ее (*слева*)

Предположим, что у нас есть простая кнопка, например как показанная на рис. 6.4, и мы хотим увеличить ее область попадания на `10px` во всех направлениях. Мы уже применили к ней некоторую стилизацию, а также добавили курсор `cursor: pointer`, который не только обеспечивает **возможность**[1] взаимодействия с помощью мыши, но и помогает проверить, где действительно начинается область попадания.

Самый простой способ расширить область попадания — создать прозрачную сплошную рамку, так как, в отличие от контуров и теней, взаимодействие мыши с рамками заставляет срабатывать события мыши на элементе. Например, для расширения области попадания элемента на `10px` во всех направлениях достаточно такого простого кода:

```
border: 10px solid transparent;
```

Однако, как видно на рис. 6.5, это плохое решение, так как оно заставляет нашу кнопку увеличиваться! Причина кроется в том, что фоны по умолчанию

[1] В сфере удобства использования термин «возможность» (*affordance*) обозначает, что элемент управления дает **очевидную видимую подсказку о том, каким образом мы можем с ним взаимодействовать**. Например, объемный вид кнопки подсказывает, что на кнопку можно нажать, а вид дверной ручки намекает, что за нее можно потянуть или повернуть ее. Подробнее об этом вы можете прочитать в статье https://ru.wikipedia.org/wiki/Возможности (или ее английском варианте https://en.wikipedia.org/wiki/Affordance). Среди профессионалов не прекращаются дискуссии, следует ли считать изменение формы указателя мыши возможностью или визуальным ответом.

растягиваются на размер рамки. Старое доброе свойство background-clip помогает ограничить фон предназначенным специально для него пространством:

```
border: 10px solid transparent;
background-clip: padding-box;
```

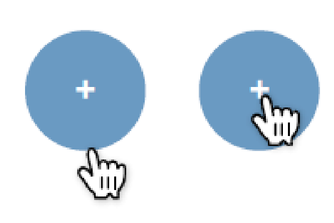


Рис. 6.5. Ой! Увеличив область попадания с помощью border, мы также сделали крупнее саму кнопку

Как подтверждает рис. 6.6, это решение прекрасно работает. Но только до тех пор, пока у вас не возникнет необходимость создать настоящую рамку вокруг кнопки и вы не поймете, что уже использовали единственную доступную вам рамку для расширения области попадания. Что же делать? Все просто: вы можете имитировать (сплошную) рамку с помощью внутренней тени (рис. 6.7):

```
border: 10px solid transparent;
box-shadow: 0 0 0 1px rgba(0,0,0,.3) inset;
background-clip: padding-box;
```

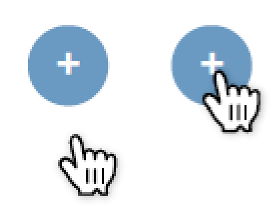


Рис. 6.6. Возвращаем нашей кнопке нормальный размер с помощью background-clip

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/hit-area-border

В отличие от рамок, при использовании box-shadow вовсе не обязательно ограничиваться одной тенью, так что если вам требуется больше, то просто перечислите наборы параметров для нужных теней через запятую. В то же время, сочетая внутренние и внешние тени, мы получаем очень странный эффект, потому что **внешние тени рисуются за пределами поля рамки**. Например, вполне логично попробовать что-то вроде решения из следующего фрагмента кода, для того чтобы добавить настоящую размытую тень, заставляющую кнопку «выпирать» из страницы (еще одна деталь, указывающая на то, что на кнопке можно щелкнуть):

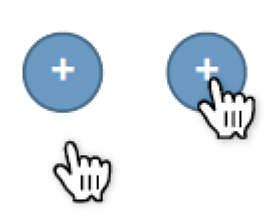


Рис. 6.7. Использование внутренней тени box-shadow для имитации рамки

```
box-shadow: 0 0 0 1px rgba(0,0,0,.3) inset,
            0 .1em .2em -.05em rgba(0,0,0,.5);
```

Но если применить эту стилизацию, результат будет совершенно не похож на ожидаемый (рис. 6.8). Это




Рис. 6.8. Добавление настоящей тени плохо работает с этим решением

решение далеко от идеала и по другим причинам. Рамки влияют на разметку, что в определенных случаях может быть неприемлемо. Что же делать? Можно удалить рамку и воспользоваться преимуществом того факта, что **псевдоэлементы также захватывают взаимодействия с мышью, определенные для их родительских элементов**.

Затем мы можем наложить на нашу кнопку прозрачный псевдоэлемент, превышающий ее по размеру на `10px` в каждом направлении:

```
button {
    position: relative;
    /* [остальные стили] */
}

button::before {
    content: '';
    position: absolute;
    top: -10px; right: -10px;
    bottom: -10px; left: -10px;
}
```

Это прекрасно работает, и пока ни один из псевдоэлементов больше никак не используется, эта кнопка ничему не мешает. Решение с псевдоэлементом необычайно гибкое — мы могли бы создать область попадания любого размера и формы и в любом месте, где только пожелаем, даже в стороне от самого элемента!

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**hit-area**

СВЯЗАННЫЕ СПЕЦИФИКАЦИИ

CSS Backgrounds & Borders: http://w3.org/TR/css-backgrounds

31 Уникальные флажки

Проблема

Дизайнеры всегда хотят иметь полный контроль над всеми элементами веб-страницы. Когда графического дизайнера, имеющего ограниченный опыт работы с CSS, просят создать макет веб-сайта, он практически всегда создает макет с уникальными стилизованными элементами управления форм, чем вгоняет в тоску разработчика, задача которого — перевести этот рисунок на язык CSS.

Для удобства чтения мы в этом секрете оперируем понятием «флажок» (*checkbox*), но это описание в равной степени применимо как к флажкам, так и к переключателям (*radio button*), если только явно не указано иное.

Когда каскадные таблицы стилей CSS только появились, предлагаемая стилизация форм была очень ограниченной, и по сей день четкого определения этих элементов ни в одной из многочисленных спецификаций CSS не существует. Однако с годами браузеры получили больше свободы в том, какие свойства CSS допускается использовать с элементами управления форм, благодаря чему мы теперь в вопросах стилизации большинства из них можем чувствовать себя довольно вольготно.

К сожалению, **флажки и переключатели** не относятся к вышеупомянутому большинству. По сей день большая часть браузеров допускает лишь **минимальную** их стилизацию или вообще **не поддерживает** стили для этих элементов форм. В результате разработчикам приходится либо мириться с их представлением по умолчанию, либо прибегать к ужасным трюкам, затрудняющим доступ к содержимому веб-страниц, таким как воссоздание этих элементов с помощью блоков `div` и сценариев JavaScript.

Существует ли способ обойти подобные ограничения и настроить внешний вид флажков, не раздувая код и не жертвуя семантикой и доступностью содержимого?

СОВЕТ

Задаетесь вопросом, в чем отличие псевдокласса `:checked` от селектора по атрибутам `[checked]`? Последний не обновляется в результате взаимодействия с пользователем, так как взаимодействие с пользователем не способно повлиять на атрибут HTML.

Решение

До недавнего времени решить эту задачу без помощи сценариев было невозможно. Однако в **Selectors Level 3** (http://w3.org/TR/css3-selectors) мы получили новый псевдокласс: `:checked`. Этот псевдокласс успешно проходит проверку на соответствие только в том случае, когда флажок отмечен — либо пользователем, либо посредством сценария.

Он не слишком полезен в применении непосредственно к флажкам, поскольку, как мы уже упоминали выше, с флажками можно использовать не так много свойств CSS. Однако мы всегда можем **прибегнуть к помощи комбинаторов для стилизации других элементов** в зависимости от состояния флажка.

Вложив флажок внутрь тега метки, мы бы смогли избавиться от необходимости использовать идентификаторы, но в этом случае у нас не было бы возможности обращаться к конкретной метке в зависимости от состояния флажка, так как родительских селекторов у нас пока что нет.

Возможно, вы спрашиваете себя, стили каких других элементов мы можем захотеть менять в зависимости от того, отмечен флажок или нет. Что ж, существует тип элемента, демонстрирующий особое поведение в привязке к флажкам, и это `<label>`. **Элемент `<label>`, связанный с флажком, также выполняет функцию переключателя для этого флажка**.

Поскольку метки (*label*), в отличие от флажков, не являются подменными элементами,[1] мы можем **добавлять к ним генерируемое содержимое и настраивать их стили в зависимости от состояния флажков**. Таким образом, можно **скрыть настоящие флажки**, сделав это так, чтобы не нарушить порядок табуляции, и вместо этого **заставить генерируемое содержимое играть роль стилизованных флажков**!

Давайте посмотрим на это решение в действии. Начнем со следующей простой разметки:

HTML

```
<input type="checkbox" id="awesome" />
<label for="awesome">Awesome!</label>
```

[1] Из спецификации CSS 2.1: «[Подменный элемент — это] элемент, содержимое которого выходит за рамки модели форматирования CSS, например изображение, встраиваемый документ или апплет». К подменным элементам невозможно применять генерируемое содержимое, хотя некоторые браузеры и поддерживают такую функциональность.

Следующий шаг заключается в генерировании псевдоэлемента, который будет использоваться в качестве нашего стилизованного флажка, и его простейшей стилизации:

```
input[type="checkbox"] + label::before {
    content: '\a0'; /* неразрывный пробел */
    display: inline-block;
    vertical-align: .2em;
    width: .8em;
    height: .8em;
    margin-right: .2em;
    border-radius: .2em;
    background: silver;
    text-indent: .15em;
    line-height: .65;
}
```



Рис. 6.9. Наш рудиментарный уникальный флажок рядом с оригинальным флажком

Стиль, который мы создаем для наших флажков в этих примерах, чрезвычайно прост, однако реальные возможности бесконечны. Вы можете даже вообще отказаться от стилизации средствами CSS и использовать изображения для всех возможных состояний флажков!

На рис. 6.9 показано, как сейчас выглядят наш флажок и метка. Оригинальный флажок все еще отображается на экране, но позднее мы скроем его. Теперь нам нужно создать другой стиль, который будет применяться к флажку, когда тот отмечен. Для начала хватит другого цвета и символа галочки в качестве содержимого:

```
input[type="checkbox"]:checked + label::before {
    content: '\2713';
    background: yellowgreen;
}
```



Рис. 6.10. Стилизация нашего псевдоэлемента под уникальный отмеченный флажок

Как видно на рис. 6.10, это решение уже функционирует как рудиментарный стилизованный флажок. Теперь необходимо спрятать оригинальный флажок, но сделать это таким образом, чтобы не нарушить доступность содержимого веб-страницы. Это означает, что мы не можем использовать `display: none`, поскольку в этом случае флажок полностью пропадет из порядка табуляции. Вместо этого применим следующий подход:

```
input[type="checkbox"] {
    position: absolute;
    clip: rect(0,0,0,0);
}
```



Будьте осторожны с подобными разрешающими селекторами. Использование `input[type="checkbox"]` **приводит к тому, что флажки, к которым не привязаны метки** (например, вложенные в тег метки), **также пропадают**, что, по сути, делает их абсолютно непригодными к использованию.







Рис. 6.11. Сверху вниз: уникальный флажок, когда он находится в фокусе; уникальный флажок, когда он недоступен; отмеченный уникальный флажок

Вот и всё! Мы сделали простейший уникальный флажок. Разумеется, мы могли бы продолжить совершенствовать его внешний вид: например, настроив разные стили для состояний, когда он находится в фокусе и когда он недоступен, как показано на рис. 6.11:

```
input[type="checkbox"]:focus + label::before {
    box-shadow: 0 0 .1em .1em #58a;
}

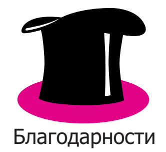
input[type="checkbox"]:disabled + label::before {
    background: gray;
    box-shadow: none;
    color: #555;
}
```

Хотя возможности поистине бесконечны, все же избегайте стилизации флажков в форме круга: большинство пользователей ассоциируют круглые элементы управления с переключателями (*radio button*). То же самое можно сказать и о квадратных переключателях.

Эти эффекты можно сделать еще привлекательнее, добавив переходы или анимации. Или же можно вообще пуститься во все тяжкие и создать какие-нибудь скевоморфные переключатели. Возможностям действительно нет предела!

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**checkboxes**


Благодарности

Благодарю Райана Седдона (Ryan Seddon) за воплощение первой версии этого эффекта, ныне известного как «трюк с флажком» (http://thecssninja.com/css/custom-inputs-using-css). С тех пор Райан успел применить эту идею для создания всевозможных виджетов, требующих сохранения состояния (http://labs.thecssninja.com/bootleg), таких как модальные диалоговые окна, раскрывающиеся меню, вкладки и карусели. Нужно только отметить, что такая интенсивная эксплуатация флажков приводит к проблемам с доступностью содержимого веб-страницы.

Переключаемые кнопки

Вариацию «трюка с флажком» можно использовать для имитации переключаемых кнопок, ведь в чистом HTML способа создавать подобные кнопки не предусмотрено. Переключаемые кнопки — это кнопки, работающие по принципу флажков: они включают или выключают настройку и выглядят нажатыми, когда настройка включена, или «отжатыми», когда настройка отключена.

Семантически никакой разницы между переключаемыми кнопками и флажками нет, так что применение данного трюка нисколько не нарушает семантическую чистоту кода.

Для того чтобы создать переключаемые кнопки с помощью данного трюка, необходимо всего лишь применить к меткам стилизацию, превращающую их в кнопки, вместо использования псевдоэлемента. Например, следующий код создает переключаемые кнопки, показанные на рис. 6.12:

```
input[type="checkbox"] {
    position: absolute;
    clip: rect(0,0,0,0);
}

input[type="checkbox"] + label {
    display: inline-block;
    padding: .3em .5em;
    background: #ccc;
    background-image: linear-gradient(#ddd,
#bbb);
    border: 1px solid rgba(0,0,0,.2);
    border-radius: .3em;
    box-shadow: 0 1px white inset;
    text-align: center;
    text-shadow: 0 1px 1px white;
}

input[type="checkbox"]:checked + label,
input[type="checkbox"]:active + label {
    box-shadow: .05em .1em .2em rgba(0,0,0,.6) inset;
    border-color: rgba(0,0,0,.3);
    background: #bbb;
}
```

Awesome!

Awesome!

Рис. 6.12. Переключаемая кнопка в обоих своих состояниях

Однако не забывайте об осторожности при использовании переключаемых кнопок. Очень часто **переключаемые кнопки снижают удобство в использовании**, так как их легко перепутать с обычными кнопками, которые по нажатии производят какое-либо действие.

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**toggle-buttons**

СВЯЗАННЫЕ СПЕЦИФИКАЦИИ

Selectors: http://w3.org/TR/selectors

32 Ослабление значимости путем затемнения

ПРЕДВАРИТЕЛЬНЫЕ ТРЕБОВАНИЯ

Знание цветов RGBA

Проблема

Нередко возникает необходимость затемнить все содержимое позади элемента с помощью полупрозрачной темной подложки, для того чтобы подчеркнуть данный элемент пользовательского интерфейса и привлечь к нему внимание пользователя. Например, этот эффект часто используется для создания «световых коробов» (рис. 6.13) и «экскурсий» по интерфейсу. В самой распространенной технике реализации роль затемняющего занавеса играет новый элемент HTML, к которому применено немного стилизации CSS, как в следующем фрагменте кода:

```
.overlay { /* Для затемнения */
    position: fixed;
    top: 0;
    right: 0;
    bottom: 0;
    left: 0;
    background: rgba(0,0,0,.8);
}

.lightbox { /* Элемент, к которому мы хотим привлечь внимание */
    position: absolute;
    z-index: 1;
    /* [остальные стили] */
}
```

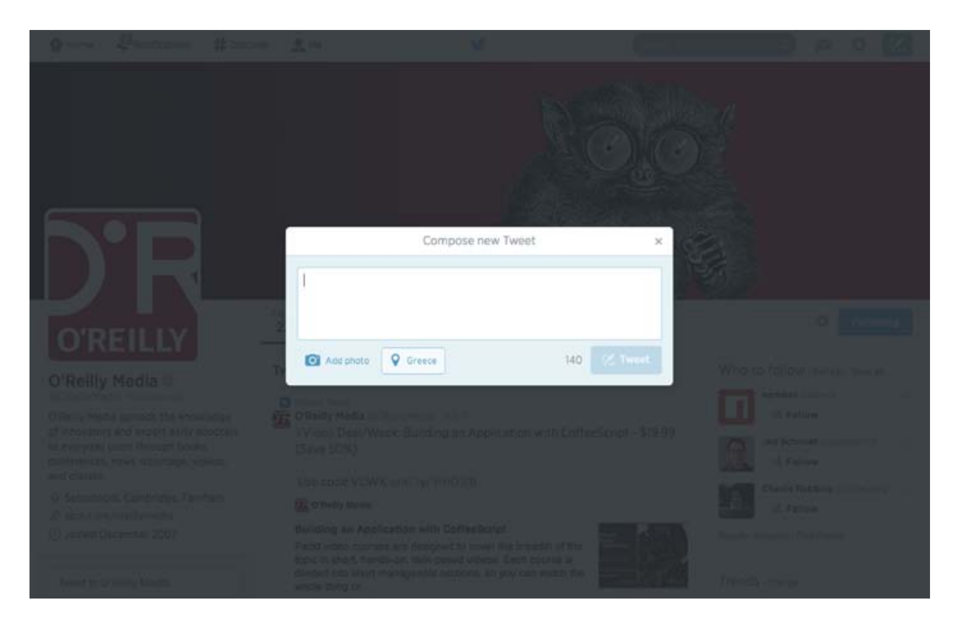


Рис. 6.13. В Twitter данный эффект используется для оформления всплывающих диалоговых окон

Подложка затеняет все содержимое позади того элемента, к которому мы хотим привлечь внимание. `.lightbox` получает более высокое значение `z-index`, для того чтобы этот элемент выводился поверх подложки. Это, конечно, прекрасно, но все же данное решение требует дополнительного элемента HTML, то есть реализовать его с помощью чистого CSS невозможно. Не самая большая проблема, но по возможности нам все же хотелось бы избежать этого неудобства. К счастью, в большинстве случаев это возможно.

Решение с псевдоэлементом

Мы можем использовать псевдоэлементы для устранения необходимости в дополнительном элементе HTML, например, так:

```
body.dimmed::before {
    position: fixed;
    top: 0;
    right: 0;
    bottom: 0;
    left: 0;
    z-index: 1;
    background: rgba(0,0,0,.8);
}
```

Это решение чуть лучше, так как теперь мы можем применять нужный эффект напрямую из CSS-кода. Однако его проблема в плохой переносимости, так как к элементу `<body>` уже может применяться какая-то другая стилизация посредством его псевдоэлемента `::before`. Также это означает, что для применения этого эффекта нам обычно требуется код JavaScript, реализующий класс `dimmed`.

Решить эту проблему можно было бы путем добавления подложки через собственный псевдоэлемент `::before` элемента, установив для него значение `z-index: -1;`, для того чтобы подложка отображалась под элементом. Однако, хотя это устраняет проблему переносимости, мы все же не можем полностью контролировать положение псевдоэлемента по оси Z. Он может оказаться под нашим элементом (как и требуется), но не исключено, что поверх него окажется **не только наш элемент, но и несколько его предков**.

Еще одна сложность с этим решением заключается в том, что **у псевдоэлементов не может быть собственных обработчиков событий JavaScript**. При использовании отдельного элемента для подложки мы можем связывать с ним обработчики событий, для того чтобы, например, световой короб закрывался, когда пользователь щелкает на подложке. Если же мы используем псевдоэлементы на том же элементе, который хотим визуально выделить, становится намного труднее определять, щелкает пользователь на элементе или на его подложке.

Решение с box-shadow

Решение с псевдоэлементом более гибкое и чаще всего его хватает для реализации подложки в привычном понимании этого слова. Однако для более простых сценариев использования или при разработке макетов мы также можем пользоваться преимуществом того факта, что радиус размазывания элемента `box-shadow` увеличивает его на значение, указанное для каждой стороны. Это означает, что мы можем создать очень большую тень без смещения и размытия, наспех сварганив некое подобие подложки:

```
box-shadow: 0 0 0 999px rgba(0,0,0,.8);
```

Очевидная проблема этого дешевого и сердитого решения — оно не работает при очень большом разрешении (> 2000px). Для устранения этого недостатка можно просто указать очень большое значение. Но можно также полностью избавиться от конкретных значений, прибегнув к помощи **единиц измерения окна просмотра**, которые способны гарантировать, что «подложка» **всегда** будет больше нашего окна просмотра. Так как мы не можем использовать разные значения радиуса размазывания по горизонтали и по вертикали, единица измерения окна просмотра, которую логично использовать в нашем случае, — это `vmax`. На случай, если вы не знакомы с единицей `vmax`, сообщаю, что `1vmax` эквивалентно либо `1vw`, либо `1vh`, смотря какое из этих двух значений больше.

`100vw` эквивалентно ширине окна просмотра, и, аналогично, `100vh` эквивалентно его высоте. Следовательно, минимальное значение, удовлетворяющее нашим требованиям, — `50vmax`, и его необходимо добавить с каждой стороны, чтобы итоговые габаритные размеры подложки на `100vmax` превосходили габаритные размеры нашего элемента:

```
box-shadow: 0 0 0 50vmax rgba(0,0,0,.8);
```

Это простая и быстрая в применении техника, но с ней связаны две довольно серьезные проблемы, ограничивающие ее применение. Можете догадаться какие?

Во-первых, так как размеры нашего элемента определяются относительно размеров окна просмотра, а не страницы, **при прокрутке мы будем видеть границы подложки**, если только для элемента не установлено свойство `position: fixed;` или если страница не слишком короткая, чтобы ее можно было прокрутить. Но так как страницы могут быть *очень* длинными, не следует пытаться преодолеть это ограничение, попросту еще сильнее увеличивая радиус размазывания. Вместо этого лучше **ограничить использование данной техники элементами с фиксированным позиционированием или страницами с минимальным объемом прокрутки или вообще ее не требующих**.

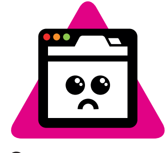
Во-вторых, использование в качестве подложки отдельного элемента (или псевдоэлемента) не только визуально направляет фокус внимания пользователя к требуемому элементу. Это также **предотвращает взаимодействие с остальными элементами страницы посредством мыши, так как события указателя мыши захватываются подложкой**. Свойство `box-shadow` такой возможности не предлагает. Следовательно, **оно только визуально притягивает внимание пользователя к определенному элементу, но само по себе не захватывает никакое взаимодействие с мышью.** Подходит это вам или нет — зависит от вашего конкретного сценария использования.

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**dimming-box-shadow**

Решение с задним фоном

Если элемент, к которому вы желаете привлечь внимание пользователя, — модальное диалоговое окно (элемент `<dialog>`, отображаемый посредством его метода `showModal()`), то у него уже есть подложка, определенная в таблице стилей User Agent. К этой родной подложке также можно добавить стили через псевдоэлемент `::backdrop`, например, чтобы сделать ее темнее:



Ограниченная поддержка

```
dialog::backdrop {
    background: rgba(0, 0, 0, .8);

}
```

Единственный недостаток этого метода заключается в том, что на момент написания данной главы он **практически не поддерживается браузерами**, поэтому, прежде чем применять его, проверьте актуальный уровень поддержки. Помните, однако, что даже если задний фон не поддерживается, то ничего не сломается, так как это всего лишь усовершенствование восприятия пользователем — просто у диалогового окна не будет подложки.

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**native-modal**

СВЯЗАННЫЕ СПЕЦИФИКАЦИИ

CSS Values & Units: http://w3.org/TR/css-values/#viewport-relativelengths

CSS Backgrounds & Borders: http://w3.org/TR/css-backgrounds

Fullscreen API: http://fullscreen.spec.whatwg.org/#::backdrop-pseudo-element

33 Ослабление значимости путем размытия

ПРЕДВАРИТЕЛЬНЫЕ ТРЕБОВАНИЯ

Переходы, **секрет «Эффект матированного стекла», секрет «Ослабление значимости путем затемнения»**

Проблема

В **секрете «Ослабление значимости путем затемнения»** мы познакомились со способом отвлечения внимания от фрагментов веб-приложения посредством их затемнения с помощью полупрозрачной черной подложки. Но если страница содержит большое количество деталей, то затемнять ее приходится очень сильно, для того чтобы обеспечить достаточный контраст с отображающимся поверх текстом или привлечь внимание к световому коробу или другому элементу. Более элегантный способ, показанный на рис. 6.14, заключается в том, что мы размываем все остальное, за исключением подсвеченного элемента, в дополнение к затемнению или вместо него. Кроме того, это создает более реалистичный эффект глубины, имитируя то, **как наше зрение воспринимает объекты, находящиеся физически ближе к нам, когда мы на них фокусируемся**.

Однако реализовать этот эффект куда сложнее. До появления спецификации **Filter Effects** (http://w3.org/TR/filter-effects) это было вообще невозможно, и даже с использованием фильтра `blur()` задача остается непростой. К чему привязывать размывающий фильтр? Или мы должны применить его ко всему, *за исключением* определенного элемента? Если мы применим его к элементу `<body>`, то будет размыто все содержимое страницы, включая элемент, к которому мы хотим привлечь внимание. Ситуация очень похожа на ту, которую мы рассматривали в **секрете «Эффект матированного стекла»**, однако прибегнуть

к тому же решению здесь мы не можем, так как позади нашего диалогового окна может находиться все что угодно, а не только фоновое изображение. Что же делать?

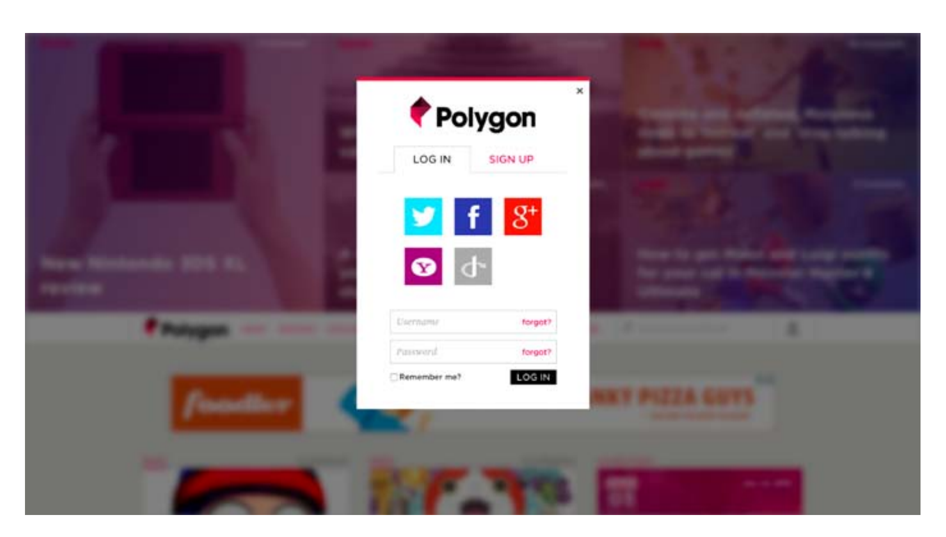
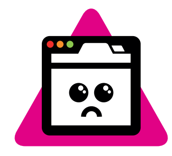


Рис. 6.14. На игровом веб-сайте polygon.com можно найти превосходный пример привлечения внимания пользователя к диалоговому окну путем размытия всего остального содержимого позади

Решение



Ограниченная поддержка

К сожалению, для данного эффекта нам потребуется дополнительный элемент HTML: мы должны будем обернуть все содержимое нашей страницы, за исключением элементов, которые не должны размываться, в элемент-обертку, а затем применить размытие к нему. Для этого идеально подойдет элемент `<main>`, так как он имеет двойное предназначение: отмечает собой основное содержимое страницы (диалоговые окна к основному содержимому обычно не относятся) и дает нам крючок, на который мы сможем навесить нужные стили. Разметка будет выглядеть приблизительно так:

HTML

```
<main>Bacon Ipsum dolor sit amet…</main>
<dialog>
    O HAI, I'm a dialog. Click on me to dismiss.
</dialog>
<!-- любые другие диалоговые окна -->
```

На рис. 6.15 вы видите, как это выглядит без подложки. Таким образом, нам необходимо применять класс к элементу `<main>` каждый раз, когда диалоговое окно отображается на экране, одновременно применяя размывающую фильтрацию, вот так:

Мы предполагаем, что все наши элементы `<dialog>` изначально скрыты и в любой момент времени на экране отображается максимум один из них.

```
main.de-emphasized {
    filter: blur(5px);
}
```

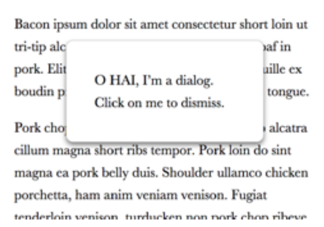


Рис. 6.15. Обычное диалоговое окно без подложки, призванной снижать значимость остального содержимого страницы

Как подтверждает рис. 6.16, это уже огромный шаг вперед. Однако сейчас размытие применяется немедленно, что выглядит не слишком естественно и ухудшает впечатление пользователя от взаимодействия со страницей. Поскольку **фильтры CSS поддерживают анимацию**, мы можем заставить размытие страницы проявляться плавно и постепенно:

```
main {
    transition: .6s filter;
}

main.de-emphasized {
    filter: blur(5px);
}
```



Рис. 6.16. Размытие элемента <main>, когда диалоговое окно отображается на экране

Часто бывает полезно комбинировать два эффекта снижения значимости (затемнение и размытие). Один из способов сделать это — использовать фильтры `brightness()` и/или `contrast()`:

```
main.de-emphasized {
    filter: blur(3px) contrast(.8)
brightness(.8);
}
```

Результат вы видите на рис. 6.17. Затемнение посредством фильтров CSS означает, что если они не поддерживаются, то **никакое резервное решение не применяется**. Возможно, затемнение лучше воплощать с помощью какого-нибудь другого метода, который также может служить резервным решением (например, используя свойство `box-shadow`, как мы делали в предыдущем секрете). Это также избавит

нас от «эффекта сияния», который можно наблюдать по краям на рис. 6.17. Обратите внимание, что на рис. 6.18, где мы использовали для затемнения тень, этой проблемы не возникает.

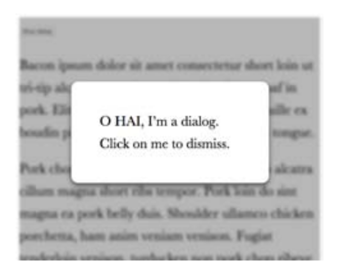


Рис. 6.17. Одновременное размытие и затемнение посредством фильтров CSS

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**deemphasizing-blur**



Рис. 6.18. Размытие с помощью фильтра CSS и затемнение посредством box-shadow, что также служит резервным решением



Благодарности

Благодарю Хакима Эль Хаттаба (Hakim El Hattab, http://hakim.se) за публикацию описания **схожего эффекта** (http://lab.hakim.se/avgrund). Кроме того, в версии эффекта, предложенной Хакимом, содержимое также уменьшается благодаря применению трансформации `scale()`, что дополнительно поддерживает иллюзию диалогового окна, находящегося физически ближе к нам, чем остальное содержимое страницы.

СВЯЗАННЫЕ СПЕЦИФИКАЦИИ

Filter Effects: http://w3.org/TR/filter-effects

CSS Transitions: http://w3.org/TR/css-transitions

34 Подсказки о прокрутке

ПРЕДВАРИТЕЛЬНЫЕ ТРЕБОВАНИЯ

Градиенты CSS, свойство `background-size`

Проблема

Полоса прокрутки — главный способ указать, что элемент включает больше содержимого, чем видно на экране. Однако очень часто они бывают неуклюжими и отвлекают внимание пользователя, поэтому разработчики современных операционных систем стремятся их упрощать, зачастую даже полностью скрывают полосы прокрутки с экрана до тех пор, пока пользователь не начинает активно взаимодействовать с элементом, поддерживающим прокрутку.

Хотя сегодня полосы прокрутки используются все реже (пользователи обычно прокручивают содержимое с помощью жестов), указание на то, что внутри элемента больше содержимого, чем видно в данный момент на экране, — чрезвычайно полезная информация, и ее рекомендуется ненавязчиво добавлять даже к тем элементам, с которыми пользователь в настоящее время не взаимодействует.

Дизайнеры-проектировщики пользовательского взаимодействия, разрабатывавшие Google Reader, клиент для чтения RSS-лент от Google (сейчас этот проект уже закрыт), нашли очень элегантный способ указания на наличие дополнительного

Ada Catlace
Alan Purring
Schrödingcat
Tim Purrners-Lee
WebKitty

Рис. 6.19. У этого поля больше содержимого, чем видно сейчас, и его можно прокрутить, но пока вы не начнете с ним взаимодействовать, вы об этом не узнаете

содержимого: если элемент содержит больше данных, чем видно на экране, сверху и/или снизу врезки отображается легкая тень (рис. 6.20).

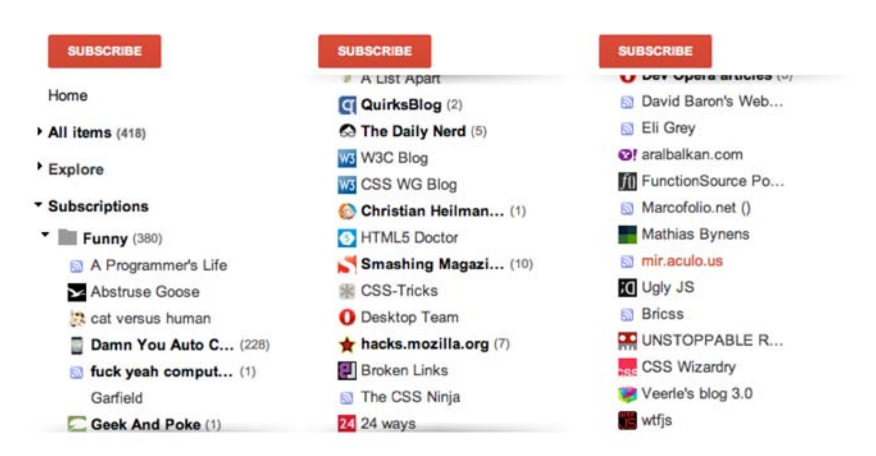


Рис. 6.20. Элегантный шаблон пользовательского взаимодействия в Google Reader, указывающий на необходимость прокрутки для просмотра полного содержимого врезки. *Слева*: содержимое прокручено до самого верха. *В центре*: содержимое прокручено до середины RSS-ленты. *Справа*: содержимое прокручено до самого низа

Однако для того чтобы добиться этого эффекта в Google Reader, разработчикам пришлось добавить немало сценариев. Было ли это действительно необходимо или тот же эффект можно реализовать на чистом CSS?

Решение

Начнем с самой простой разметки, обычного неупорядоченного списка с бессмысленным содержимым (эксцентричными кличками для кошек!):

HTML

```
<ul>
    <li>Ada Catlace</li>
    <li>Alan Purring</li>
    <li>Schrödingcat</li>
    <li>Tim Purrners-Lee</li>
    <li>WebKitty</li>
    <li>Json</li>
    <li>Void</li>
    <li>Neko</li>
    <li>NaN</li>
    <li>Cat5</li>
    <li>Vector</li>
</ul>
```

Теперь мы можем применить к элементу <ul> простейшую стилизацию, для того чтобы сделать его меньше длины его содержимого и добавить возможность прокрутки:

```
overflow: auto;
width: 10em;
height: 8em;
padding: .3em .5em;
border: 1px solid silver;
```

Здесь и начинается самое интересное. Давайте создадим наверху тень с помощью радиального градиента:

```
background: radial-gradient(at top, rgba(0,0,0,.2),
                            transparent 70%) no-repeat;
background-size: 100% 15px;
```

Результат вы видите на рис. 6.21. Пока что эта тень остается на одном месте, даже если мы прокручиваем содержимое. Это соответствует тому, как фоновые изображения работают по умолчанию: их позиция всегда зафиксирована относительно элемента, независимо от того, насколько сильно мы прокручиваем содержимое элемента. Это правило распространяется также и на изображения со свойством `background-attachment: fixed`; единственное отличие в том, что они также **остаются на своем месте, когда пользователь прокручивает содержимое страницы.** Можно ли заставить фоновое изображение прокручиваться вместе с содержимым элемента?

Рис. 6.21. Тень наверху элемента

До недавнего времени реализовать этот простой эффект было невозможно. Однако наличие проблемы было очевидно, и для ее решения в **Backgrounds & Borders Level 3** (http://w3.org/TR/css3-background/#local0) для `background-attachment` было добавлено новое ключевое слово: `local`.

Но `background-attachment: local` не решает нашу проблему без дополнительной обработки напильником. Если мы применим это свойство к нашей градиентной тени, то результат будет прямо противоположным: тень будет отображаться, когда содержимое прокручено до самого верха, а при прокрутке содержимого вниз тень будет пропадать. Но для начала уже неплохо — нужно же с чего-то начинать.

Секрет трюка в том, чтобы использовать два фона: один для тени, а второй — представляющий собой, по сути, **белый прямоугольник, закрывающий тень**

и играющий роль маски. Для фона, генерирующего тень, будет установлено значение background-attachment по умолчанию (scroll), так как мы хотим, чтобы он всегда оставался на своем месте. Однако для маскирующего фона мы установим значение свойства background-attachment, равное local, для того чтобы он закрывал тень, когда содержимое прокручивается до самого верха. Когда же мы будем прокручивать содержимое вниз, он будет прокручиваться вместе с содержимым, открывая таким образом тень.

Для создания маскирующего прямоугольника мы воспользуемся линейным градиентом того же цвета, что и фоновый цвет элемента (в нашем случае это белый):

```
background: linear-gradient(white, white),
            radial-gradient(at top, rgba(0,0,0,.2),
                            transparent 70%);
background-repeat: no-repeat;
background-size: 100% 15px;
background-attachment: local, scroll;
```

На рис. 6.22 вы видите, как это решение работает на разных этапах прокрутки. Очевидно, что нам удалось добиться желаемого эффекта, но не без одного большого недостатка: когда мы только начинаем прокручивать список, тень открывается странным образом и поначалу выглядит обрезанной. Можно ли сделать эффект более гладким и естественным?

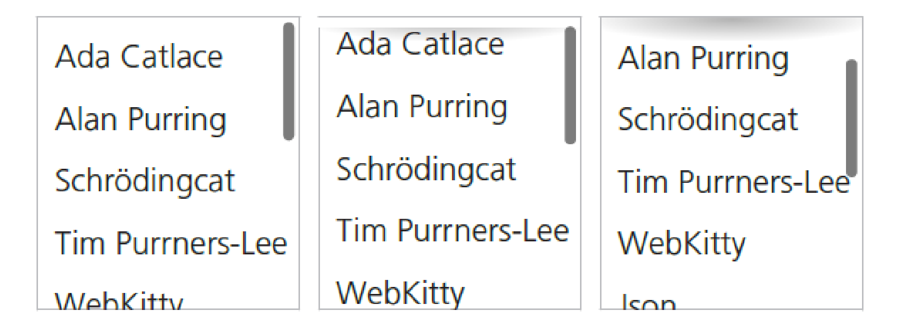


Рис. 6.22. Наши два фона на разных этапах прокрутки. *Слева*: список прокручен до самого верха. *Посередине*: совсем небольшая прокрутка вниз. *Справа*: список прокручен вниз в значительной степени

Мы можем воспользоваться преимуществом того факта, что наша «маска» по природе своей — это (вырожденный) линейный градиент, и преобразовать его в настоящий градиент от цвета white до прозрачного белого (hsla(0,0%,100%,0) или rgba(255,255,255,0)), для того чтобы обеспечить плавное отображение нашей тени:

```
background: linear-gradient(white, hsla(0,0%,100%,0)),
            radial-gradient(at top, rgba(0,0,0,.2),
                            transparent 70%);
```

Рис. 6.23. Использование градиента от white до transparent в качестве первой попытки обеспечить плавный вывод тени

Это шаг в правильном направлении. Как вы видите на рис. 6.23, это обеспечивает плавное постепенное отображение тени, как мы и хотели. Однако пока что у нашего решения есть довольно серьезный недостаток: когда список прокручен до самого верха, тень теперь закрывается не полностью. Этот огрех можно исправить, сместив границу перехода цвета `white` чуть ниже (если точнее, то на `15px` — такова высота нашей тени), чтобы получить поле сплошного белого цвета, прежде чем начнется переход к прозрачности. Помимо этого, нам необходимо увеличить размер «маски», для того чтобы он был больше тени, иначе градиента не получится. Точное значение высоты зависит от того, насколько плавный эффект вы желаете реализовать (то есть насколько быстро тень должна появляться после начала прокрутки). После серии экспериментов я пришла к выводу, что `50px` — разумное значение. Финальная версия кода выглядит, как показано далее, а результат вы можете видеть на рис. 6.24:

Почему прозрачный белый, а не просто `transparent`? Второе значение — это на самом деле всего лишь псевдоним для `rgba(0,0,0,0)`, поэтому градиент может включать оттенки серого в переходах от непрозрачного белого к прозрачному черному. Если браузеры интерполируют цвета в рамках данной спецификации в так называемом *цветовом пространстве premultiplied RGBA*, то такое не должно случаться. Разные алгоритмы интерполяции не входят в список тем, рассматриваемых в этой книге, но если вам интересно, в Сети вы найдете огромное количество разнообразных материалов.

```
background: linear-gradient(white 30%, transparent),
            radial-gradient(at 50% 0, rgba(0,0,0,.2),
                            transparent 70%);
background-repeat: no-repeat;
background-size: 100% 50px, 100% 15px;
background-attachment: local, scroll;
```

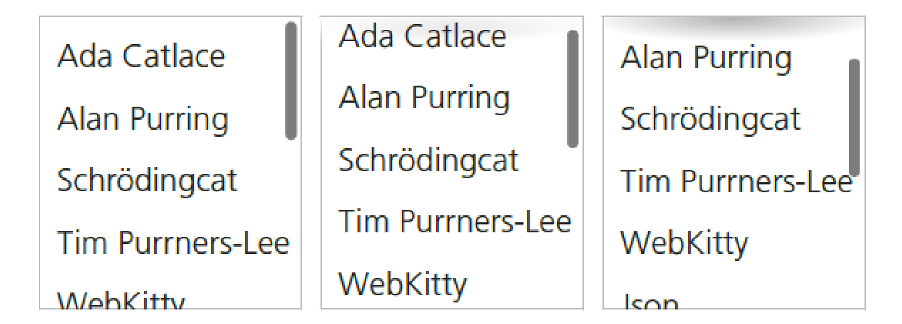
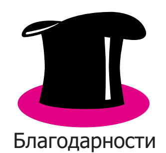


Рис. 6.24. Конечный результат

Разумеется, для того чтобы достичь исходного эффекта, нам потребуются **еще два градиента для нижней тени, а также маска для нее**, но логика остается той же самой, так что я оставлю это в качестве упражнения для читателя (или загляните в пример из врезки «Попробуйте сами!» ниже).

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**scrolling-hints**



Благодарю **Романа Комарова** за **первый пример реализации этого эффекта** (http://kizu.ru/en/fun/shadowscroll). Его версия основывается на псевдоэлементах и позиционировании, а не фоновых изображениях, и это может быть интересной альтернативой для определенных сценариев использования.

СВЯЗАННЫЕ СПЕЦИФИКАЦИИ

CSS Backgrounds & Borders: http://w3.org/TR/css-backgrounds

CSS Image Values: http://w3.org/TR/css-images

35 Интерактивное сравнение изображений

Проблема

Иногда перед вами встает задача показать визуальные различия между двумя изображениями, обычно в качестве иллюстрации «было — стало». Например, для того чтобы продемонстрировать в портфолио результаты обработки фотографии, показать эффект определенных процедур на веб-сайте салона красоты или проиллюстрировать видимые последствия катастрофического события в какой-то географической области.

Самое распространенное решение — просто поместить два изображения рядом друг с другом. Однако при этом человеческий глаз замечает только очевидные различия и упускает мелочи. Это не страшно, если детали не так важны или же различия действительно очень значительные, но во всех остальных случаях нам требуется более удобный способ сравнения.

С точки зрения удобства использования у данной проблемы есть несколько решений. Одно из наиболее распространенных — показывать оба изображения в одном и том же месте, быстро сменяя одно другим, используя для этого анимированное изображение в формате GIF или анимацию CSS. Это намного лучше, чем выводить изображения подле друг друга, но пользователю приходится потратить время на то, чтобы заметить все различия, так как ему нужно просмотреть несколько итераций, каждый раз фиксируя взгляд на новой области изображения.

В некоторых вариациях пользователь всего лишь двигает указатель мыши, вместо того чтобы перетаскивать полосу. Преимущество такого подхода в том, что его проще заметить и использовать, но создаваемое мельтешение может раздражать.

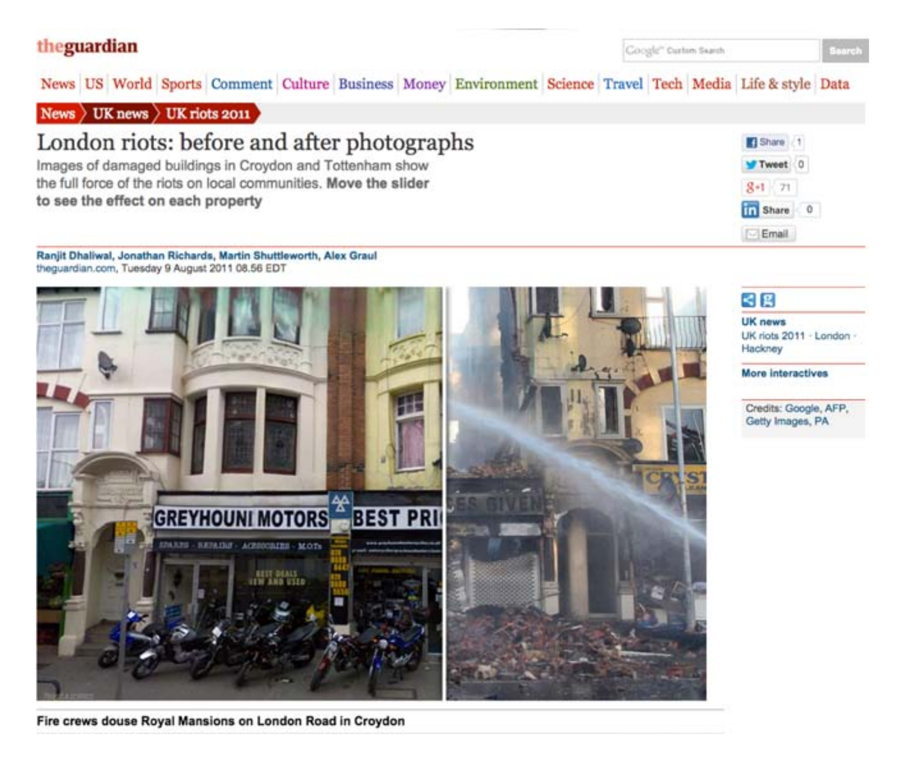


Рис. 6.25. Пример интерактивного виджета для сравнения изображений, позволяющего пользователям оценить катастрофические последствия беспорядков в Лондоне в 2011 году (сайт крупнейшей британской газеты The Guardian). Подразумевается, что пользователь будет перетаскивать белую полосу, разделяющую два изображения, но ничто на самом изображении не намекает на то, что полоса поддается перетаскиванию, поэтому авторам пришлось добавить текстовую подсказку (*Move the slider*...). В идеальном случае хороший, легкий в изучении интерфейс не требует наличия такого вспомогательного текста

Источник: http://theguardian.com/uk/interactive/2011/aug/09/london-riots-before-after-photographs

Намного более удобное для пользователя решение — так называемый слайдер сравнения изображений. Этот элемент управления содержит оба изображения, одно поверх другого, и позволяет пользователю перетаскивать разделитель, открывая одно или второе. Разумеется, такого элемента управления в HTML в действительности не существует. Его приходится имитировать средствами имеющихся элементов, и в Сети можно найти массу вариантов реализации, чаще всего требующих каркасов JavaScript и большого количества JS-кода.

Существует ли более простой способ добавления на страницу подобного элемента управления? Да, причем целых два!

Решение со свойством resize в CSS

Если подумать, то слайдер сравнения изображений, по сути, состоит из изображения и элемента с изменяющимся горизонтальным размером, который постепенно открывает другое изображение. Именно здесь на помощь обычно призывается каркас JavaScript: он обеспечивает возможность изменения размера верхнего изображения по горизонтали. Однако для того, чтобы сделать размер элемента динамичным, вовсе не обязательно прибегать к помощи сценариев. В **CSS User Interface Level 3** (http://w3.org/TR/css3-ui/#resize) мы получили новое свойство, предназначенное специально для выполнения этой задачи: скромное `resize`!

Даже если вы никогда не слышали о таком свойстве, то наверняка видели его в действии, так как по умолчанию для него устанавливается значение `both` для элементов `<textarea>`, благодаря чему размер текстовых полей можно менять в обоих направлениях. Но в действительности это свойство можно устанавливать для любых элементов при условии, что значение `overflow` для данного элемента **не** равно `visible`. По умолчанию значение `resize` почти для всех элементов равно `none`, что запрещает изменение их размера. Помимо `both`, это свойство также принимает значения `horizontal` и `vertical`, ограничивающие направление изменения размера.

Вы наверняка уже задаетесь вопросом, можно ли применить это свойство для реализации нашего слайдера сравнения изображений. Что ж, не узнаем, пока не попробуем!

Первой мыслью может быть всего лишь добавить два элемента `<img>`. Однако применение `resize` напрямую к `<img>` дает ужасные результаты, поскольку при изменении размера изображения оно искажается. Гораздо разумнее установить это свойство для контейнера `<div>`. В итоге мы получим примерно такую разметку:

Хорошая идея во многих ситуациях — устанавливать для `<textarea>` свойство `resize: vertical`, для того чтобы разрешить изменение размера, но только по вертикали, так как горизонтальное изменение размера обычно ломает макет страницы.

Когда `object-fit` и `object-position` будут лучше поддерживаться браузерами, это перестанет быть проблемой, так как мы сможем контролировать способ изменения размера изображений точно так же, как уже сейчас контролируем масштабирование фоновых изображений.

HTML

```
<div class="image-slider">
    <div>
        <img src="adamcatlace-before.jpg" alt="Before" />
    </div>
    <img src="adamcatlace-after.jpg" alt="After" />
</div>
```

Затем применим немного простейшего CSS для позиционирования и определения размеров:

```
.image-slider {
    position:relative;
    display: inline-block;
}

.image-slider > div {
    position: absolute;
    top: 0; bottom: 0; left: 0;
    width: 50%; /* Первоначальное значение ширины */
    overflow: hidden; /* Изображение должно обрезаться */
}

.image-slider img { display: block; }
```



Рис. 6.26. После базовой стилизации это уже начинает напоминать слайдер для сравнения изображений, но пока мы не можем менять ширину верхнего изображения

Сейчас результат выглядит как на рис. 6.26, но элемент пока что статичный. Если мы вручную будем менять ширину, то сможем пронаблюдать все этапы изменения представления двух изображений. Для того чтобы ширина менялась динамически через взаимодействие с пользователем, но посредством свойства `resize`, нам нужно добавить еще два объявления:

```
.image-slider > div {
    position: absolute;
    top: 0; bottom: 0; left: 0;
    width: 50%;
    overflow: hidden;
    resize: horizontal;
}
```

Единственное визуальное отличие состоит в том, что теперь в правом нижнем углу изображения «до» отображается манипулятор для изменения размера (рис. 6.27), но теперь мы можем перетаскивать

и менять ширину изображения сколько нашей душе угодно! Однако поиграв с нашим виджетом, мы обнаруживаем несколько слабых сторон:

- изменяя ширину элемента <div>, мы можем выходить за пределы изображений;
- манипулятор для изменения размера не так просто заметить.



Рис. 6.27. Наш слайдер для сравнения изображений действительно выполняет функцию слайдера, но у текущего решения все же есть несколько недостатков

Первую проблему решить просто. Нам всего лишь нужно задать для свойства `max-width` значение `100%`. Второй вопрос несколько сложнее. К сожалению, не существует стандартного способа менять размер манипулятора. Некоторые механизмы визуализации поддерживают собственные псевдоэлементы (такие, как `::-webkit-resizer`), но результаты их применения ограничены как в терминах поддержки браузерами, так и в терминах гибкости стилизации. И все же не теряйте надежды: оказывается, если наложить псевдоэлемент на манипулятор для изменения размера, то это нисколько не повредит его функциональности, даже без `pointer-events: none`. Таким образом, решением, подходящим для разных браузеров, будет всего лишь... наложить на наш манипулятор еще один манипулятор. Давайте сделаем это:

```
.image-slider > div::before {
    content: '';
    position: absolute;
    bottom: 0; right: 0;
    width: 12px; height: 12px;
    background: white;
    cursor: ew-resize;
}
```

Обратите внимание на объявление `cursor: ew-resize` — оно дополнительно намекает на **возможность взаимодействия** с элементом, подсказывая пользователю, что здесь находится манипулятор. Однако **мы не должны надеяться на изменение внешнего вида курсора как на единственную подсказку**, поскольку они заметны, только когда пользователь уже взаимодействует с элементом управления.

Сейчас наш манипулятор для изменения размера выглядит как белый квадратик (рис. 6.28). Но мы можем сделать шаг вперед и изменить его стилизацию в соответствии с собственными вкусами. Например, давайте превратим его в белый треугольник, отстоящий от краев изображения на `5px` (как показано на рис. 6.29). Для этого нам потребуется такой код:



Рис. 6.28. Стилизация манипулятора для изменения размера под белый квадратик путем наложения на него псевдоэлемента

```
padding: 5px;
background:
    linear-gradient(-45deg, white 50%,
                   transparent 0);
background-clip: content-box;
```

В качестве дополнительного усовершенствования мы могли бы применить `user-select: none` к обоим изображениям, чтобы ошибка при захвате манипулятора не приводила к их бессмысленному выделению. В итоге окончательная версия кода будет выглядеть так:



Рис. 6.29. Стилизация псевдоэлемента, отвечающего за фальшивый манипулятор, в форме треугольника, сдвинутого на 5px относительно краев изображения

```
.image-slider {
    position: relative;
    display: inline-block;
}

.image-slider > div {
    position: absolute;
    top: 0; bottom: 0; left: 0;
    width: 50%;
    max-width: 100%;
    overflow: hidden;
    resize: horizontal;
}

.image-slider > div::before {
    content: '';
    position: absolute;
    bottom: 0; right: 0;
    width: 12px; height: 12px;
    padding: 5px;
    background:
        linear-gradient(-45deg, white 50%,
                       transparent 0);
    background-clip: content-box;
    cursor: ew-resize;
}

.image-slider img {
    display: block;
    user-select: none;
}
```

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**image-slider**

Решение с ползунком

Метод со свойством `resize` из CSS, описанный в предыдущем разделе, отлично работает и требует совсем небольшого объема кода. Однако у него есть несколько недостатков:

- им **невозможно воспользоваться при работе только с клавиатуры**;
- перетаскивание — единственный способ изменить размер верхнего изображения, что может превратиться в **утомительную** задачу, если изображение достаточно велико или если у пользователя нарушены моторные функции. Если бы у пользователя была также возможность **щелкнуть в какой-то точке** и тем самым изменить размер изображения до этой позиции, то таким виджетом пользоваться было бы намного удобнее;
- пользователь может изменить верхнее изображение только путем перетаскивания его правого нижнего угла, манипулятор в котором бывает сложно заметить, даже если мы определяем для него дополнительные стили, как описано выше.

Если вы не против некоторого объема сценариев, то мы можем воспользоваться **элементом управления** `slider` (ползунок в HTML). Мы наложим его поверх наших изображений и настроим так, чтобы он мог управлять изменением размера. Это решит все три перечисленные выше проблемы. Так как мы все равно планируем использовать JS, то можем добавить еще несколько элементов посредством сценариев, поэтому начнем с самой пустой разметки, какая только возможна:

HTML

```
<div class="image-slider">
    <img src="adamcatlace-before.jpg" alt="Before" />
    <img src="adamcatlace-after.jpg" alt="After" />
</div>
```

Затем наш код JS преобразует его в следующую версию и добавит на ползунке событие, для того чтобы он также устанавливал ширину блока `div`:

HTML

```
<div class="image-slider">
    <div>
        <img src="adamcatlace-before.jpg" alt="Before" />
    </div>
    <img src="adamcatlace-after.jpg" alt="After" />
    <input type="range" />
</div>
```

Сам код JavaScript довольно прост:

JS

```
$$('.image-slider').forEach(function(slider) {
    // Создаем дополнительный блок div и
    // оборачиваем его вокруг первого изображения
    var div = document.createElement('div');
    var img = slider.querySelector('img');
    slider.insertBefore(img, div);
    div.appendChild(img);

    // Создаем ползунок
    var range = document.createElement('input');
    range.type = 'range';
    range.oninput = function() {
        div.style.width = this.value + '%';
    };
    slider.appendChild(range);
});
```

CSS-код, который мы возьмем в качестве отправной точки, практически аналогичен версии из предыдущего решения. Мы только удалим несколько ненужных фрагментов:

- нам больше не требуется свойство `resize`;
- нам не требуется правило `.image-slider > div::before`, так как у нас больше нет манипулятора для изменения размера;
- нам не требуется свойство `max-width`, потому что этим значением будет управлять ползунок.

Вот как наш CSS-код выглядит после этих модификаций:

```
.image-slider {
    position:relative;
    display: inline-block;
}

.image-slider > div {
```

```
    position: absolute;
    top: 0; bottom: 0; left: 0;
    width: 50%;
    overflow: hidden
}
.image-slider img {
    display: block;
    user-select: none;
}
```

Если мы сейчас протестируем этот код, то увидим, что он уже работает, но результат выглядит ужасно: ползунок находится в произвольном месте под изображениями (рис. 6.30). Нам необходимо применить несколько стилей CSS, для того чтобы поместить его на изображения и совместить с ними по ширине:

```
.image-slider input {
    position: absolute;
    left: 0;
    bottom: 10px;
    width: 100%;
    margin: 0;
}
```

Как видно на рис. 6.31, результат выглядит уже вполне прилично. Существует несколько специализированных псевдоэлементов, позволяющих добавлять к ползункам желаемые стили, оформляя их по своему вкусу. Среди них `::-moz-range-track`, `::-ms-track`, `::-webkit-slider-thumb`, `::-moz-range-thumb` и `::-ms-thumb`. Но как это часто бывает со специализированными возможностями, результаты их применения непоследовательны, хрупки и непредсказуемы, поэтому я рекомендую отказаться от их использования, если только у вас нет **действительно** основательных причин внедрять их. Я свое слово сказала.

Если еще одним нашим пожеланием является некоторая визуальная унификация ползунка с основным элементом, то нам может помочь режим смешивания и/или фильтр. Режимы смешивания `multiply`, `screen` и `luminosity` дают довольно хорошие результаты. Кроме того, `filter: contrast(4)` способен сделать ползунок черно-белым, а значение контраста меньше

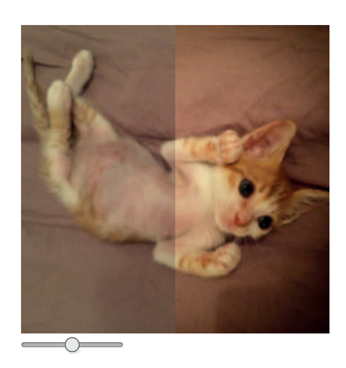


Рис. 6.30. Наш элемент управления теперь работает, но мы все еще должны создать стили для ползунка

СОВЕТ

Используйте `input:in-range` вместо простого `input`, для того чтобы стили применялись к ползунку только в том случае, если ползунки поддерживаются. Тогда благодаря каскадным свойствам вы сможете скрывать его в старых браузерах или оформлять иным способом.

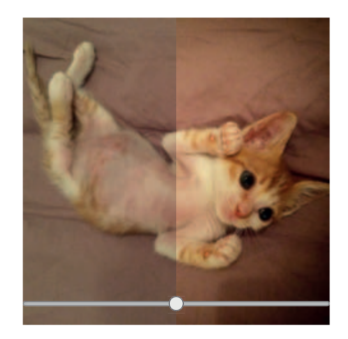


Рис. 6.31. Благодаря стилям наш ползунок теперь находится поверх изображений

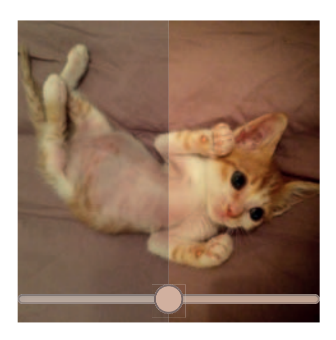


Рис. 6.32. Использование режимов смешивания и фильтров для визуальной унификации ползунка и основного элемента управления. Также мы сделали ползунок больше с помощью трансформаций CSS

единицы способно добавить оттенков серого. Возможности бесконечны, и единственно верного выбора здесь быть не может. Вы можете даже сочетать режимы смешивания и фильтры, например так:

```
filter: contrast(.5);
mix-blend-mode: luminosity;
```

Помимо этого, мы можем увеличить область, которую пользователь использует для изменения размера, для того чтобы ему было удобнее это делать (в соответствии с законом Фиттса). Для этого уменьшим значение ширины и компенсируем различие с помощью трансформаций CSS:

```
width: 50%;
transform: scale(2);
transform-origin: left bottom;
```

Результат применения обеих стилизаций вы видите на рис. 6.32. Еще одно преимущество данного подхода — хотя и временное — заключается в том, что ползунки в настоящее время поддерживаются браузерами лучше, чем свойство `resize`.



Благодарности

Спасибо **Дадли Стори** (Dudley Storey) за **первую версию этого решения** (http://demosthenes.info/blog/819/A-Before-And-After-Image-Comparison-Slide-Control-in-HTML5).

СВЯЗАННЫЕ СПЕЦИФИКАЦИИ

CSS Basic User Interface: http://w3.org/TR/css3-ui

CSS Image Values: http://w3.org/TR/css-images

CSS Backgrounds & Borders: http://w3.org/TR/css-backgrounds

Filter Effects: http://w3.org/TR/filter-effects

Compositing and Blending: http://w3.org/TR/compositing

CSS Transforms: http://w3.org/TR/css-transforms

Структура и макет



36 Определение размера изнутри

Проблема

Как мы все знаем, если мы не устанавливаем конкретное значение `height` для элемента, то его высота автоматически корректируется в зависимости от размера содержимого. А что, если нам бы хотелось наблюдать схожее поведение атрибута `width`? Например, предположим, что мы оформляем иллюстрации с помощью HTML5, используя примерно такую разметку:

HTML

```
<p>Some text […]</p>
<figure>
    <img src="adamcatlace.jpg" />
    <figcaption>
        The great Sir Adam Catlace was named after
        Countess Ada Lovelace, the first programmer.
    </figcaption>
</figure>
<p>More text […].</p>
```

Также предположим, что мы применяем какую-то простейшую стилизацию, скажем, создаем рамку вокруг иллюстраций. По умолчанию результат выглядит как на рис. 7.1. Но мы хотим, чтобы **ширина иллюстраций была равна ширине содержащихся в них изображений** (которые могут быть абсолютно разными), а также чтобы они **выравнивались по центру по горизонтали**. Текущий вариант визуализации далек от желаемого результата: строки текста намного длиннее изображения. Как же сделать так, чтобы ширина иллюстрации определялась шириной содержащегося в ней изображения, а не шириной родительского элемента?[1] За время своей карьеры вы наверняка успели составить собственный

[1] На профессиональном жаргоне CSS это означает, что ширина должна определяться *изнутри,* а не *снаружи*.

список CSS-стилей, которые приводят к такому поведению свойства `width`, чаще всего в качестве побочного эффекта:

- если мы сделаем элемент `<figure>` плавающим, то это даст нам нужную ширину, но кардинальным образом изменит макет иллюстрации, что нам, скорее всего, совершенно не нужно (рис. 7.2);
- установка свойства `display: inline-block` для иллюстрации позволяет определять ее размер в зависимости от ее содержимого, но не так, как нам бы этого хотелось (рис. 7.3). Помимо этого, даже если бы способ вычисления ширины соответствовал нашим ожиданиям, было бы чрезвычайно сложно в таких условиях выравнивать иллюстрацию по центру. Нам понадобилось бы применить `textalign: center` к ее родительскому элементу и `text-align: left` к любым другим возможным потомкам этого родительского элемента (`p`, `ul`, `ol`, `dl`, ...);
- в качестве последнего средства разработчики часто прибегают к установке фиксированного значения `width` или `max-width` для иллюстраций и применению `max-width: 100%` к `figure > img`. Однако это приводит к неэффективному расходованию доступного пространства, может давать уродливые результаты для слишком маленьких иллюстраций, а также не адаптивно.

Существует ли достойное решение этой проблемы на чистом CSS или же нам следует сдаться и приступить к кодированию сценария, динамически устанавливающего ширину иллюстраций?



Рис. 7.1. Визуализация нашей разметки по умолчанию; немного CSS-кода добавлено для создания рамок и определения ширины забивки

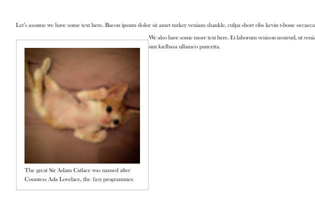


Рис. 7.2. Попытка решить проблему ширины с помощью плавающего элемента порождает новые проблемы



Рис. 7.3. Вопреки нашим ожиданиям, display: inline-block не дает нам иллюстрацию нужной ширины

Решение

Относительно новая спецификация, **CSS Intrinsic & Extrinsic Sizing Module Level 3** (http://w3.org/TR/css3-sizing), определяет несколько новых ключевых слов `width` и `height`, одним из самых полезных среди которых является `min-content`. Это ключевое слово дает нам ширину самого большого неразделяемого

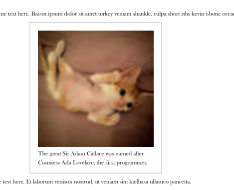


Рис. 7.4. Итоговый результат

элемента внутри поля (то есть самое широкое слово или изображение либо поле фиксированной ширины). Это в точности то, что нам требуется! Теперь задать для наших иллюстраций подходящую ширину и высоту, выровняв их по горизонтали, можно с помощью всего лишь двух строк кода:

```
figure {
    width: min-content;
    margin: auto;
}
```

Еще одно значение, `max-content`, дает нам то же значение ширины, которое мы получали с `display: inline-block` в примере выше. А `fit-content` определяет то же поведение, что и с плавающими блоками (очень часто, но не всегда совпадающее с поведением при использовании `min-content`).

Результат вы можете видеть на рис. 7.4. Для обеспечения изящного обходного пути в старых браузерах эту технику можно объединить с установкой фиксированного значения `max-width`, например так:

```
figure {
    max-width: 300px;
    max-width: min-content;
    margin: auto;
}

figure > img { max-width: inherit; }
```

В современном браузере второе определение `max-width` замещает первое, но если размер иллюстрации определяется изнутри, то `max-width: inherit` не оказывает никакого эффекта.

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**intrinsic-sizing**



Спасибо **Дадли Стори** (Dudley Storey, http://demosthenes.info) за то, что придумал этот сценарий использования (http://demosthenes.info/blog/662/Design-From-the-Inside-Out-With-CSS-MinContent).

СВЯЗАННЫЕ СПЕЦИФИКАЦИИ

CSS Intrinsic & Extrinsic Sizing: http://w3.org/TR/css3-sizing

37 Укрощение ширины столбцов таблиц

Проблема

Хотя мы уже очень давно прекратили использовать таблицы для создания макетов, они все же занимают достойное место на современных веб-сайтах: они используются для отображения таких табличных данных, как статистика, сообщения электронной почты, перечисления элементов с большим объемом метаданных и многого другого. Кроме того, мы можем заставлять другие элементы демонстрировать свойства, характерные для таблицы, используя ключевые слова свойства `display`, связанные с таблицами. Однако каким бы удобным инструментом они ни были в определенных обстоятельствах, макет таблиц ведет себя очень непредсказуемо, когда дело доходит до динамического содержимого. Причина этого в том, что размеры столбцов корректируются в зависимости от объема содержимого, и даже явные объявления `width` считаются не более чем подсказками, как видно на рис. 7.5.

По этой причине нам часто приходится либо использовать другие элементы для отображения табличных данных, либо мириться с непредсказуемостью макета. Существует ли способ заставить таблицы вести себя прилично?

Решение

Решение приходит в форме малоизвестного **свойства CSS 2.1** под названием `table-layout`. Его значение по умолчанию равно `auto`, что определяет так называемый *алгоритм автоматического расчета табличного макета*, демонстрирующий знакомое поведение, показанное на рис. 7.5. Однако у него есть и второе значение, `fixed`, обеспечивающее **более предсказуемое поведение**. Оно дает

Если мы не…	указываем ширину ячеек, то им присваиваются значения ширины, зависящие от их содержимого. Обратите внимание, что здесь ячейка с большим объемом содержимого намного шире.

Если мы не…	указываем ширину ячеек, то им присваиваются значения ширины, зависящие от их содержимого. Обратите внимание, что здесь ячейка с большим объемом содержимого намного шире.
Все строки учитываются при вычислении значений ширины, а не только первая.	Обратите внимание, что здесь размеры ячеек отличаются от предыдущего примера.

Даже если мы задаем значение ширины, то не всегда получаем его на выходе. Моя ширина равна `1000px`…	… а моя ширина равна `2000px`. Но так как здесь недостаточно места для `3000px`, ячейки пришлось пропорционально уменьшить до 33,3% и 66,6% общей ширины соответственно.

Если мы запретим перенос строк, то таблица может стать такой большой, что вылезет за пределы своего контейнера.	…и `text-overflow: ellipsis` не поможет.

Большие изображения и блоки кода также могут привести к возникновению этой проблемы.	

Рис. 7.5. Алгоритм разметки таблицы по умолчанию для таблиц с двумя столбцами и содержимым переменного размера (контейнер этих таблиц обозначен пунктирной рамкой)

Если мы не…	указываем ширину ячеек, то им присваиваются значения ширины, зависящие от их содержимого. Обратите внимание, что здесь ячейка с большим объемом содержимого намного шире.

Если мы не…	указываем ширину ячеек, то им присваиваются значения ширины, зависящие от их содержимого. Обратите внимание, что здесь ячейка с большим объемом содержимого намного шире.
Все строки учитываются при вычислении значений ширины, а не только первая.	Обратите внимание, что здесь размеры ячеек отличаются от предыдущего примера.

Даже если мы задаем значение ширины, то не всегда получаем его на выходе. Моя ширина равна `1000px`…

Если мы запретим перенос строк, то таблица может стать такой большой, что вылезет за пределы своего контейнера.	…и `text-overflow: ellipsis` не п.

Большие изображения и блоки кода также могут привести к возникновению этой проблемы.	

Рис. 7.6. Те же таблицы, что и на рис. 7.5, но с установленным свойством table-layout: fixed. Обратите внимание на следующие особенности (для каждой из показанных таблиц):

- когда мы не определяем никаких значений ширины, ширина всех столбцов становится одинаковой;
- вторая строка никак не влияет на ширину столбцов;
- если задано большое значение ширины, то оно применяется как есть и столбцы не сжимаются;
- свойства overflow и text-overflow работают как задумано и не игнорируются;
- содержимое может вытекать за пределы ячеек таблицы (если для свойства overflow установлено значение visible)

большую свободу действий разработчику (да, вам!), снимая часть ответственности с механизма визуализации. Стили действительно принимаются во внимание и не считаются простыми подсказками, переливание через край происходит так же, как с любым другим элементом (включая `text-overflow`), а содержимое таблицы влияет только на высоту каждой строки и больше ни на что.

Помимо лучшей предсказуемости и удобства, **алгоритм фиксированного табличного макета работает значительно быстрее**. Поскольку содержимое таблицы не влияет на ширину ячеек, никакие элементы во время загрузки страницы не перерисовываются. Всем нам знакома ситуация, когда по мере загрузки страницы таблица постоянно перерисовывается из-за изменения ширины столбцов. С фиксированными табличными макетами об этом можно забыть.

Чтобы воспользоваться этой возможностью, необходимо установить данное свойство для элементов `<table>` и элементов со свойством `display: table`. Обратите внимание, что для того, чтобы фокус сработал, необходимо обязательно задать значение ширины для этих таблиц (даже если оно равно `100%`). Кроме того, для того чтобы работало свойство `text-overflow: ellipsis`, следует также задать ширину соответствующего столбца. Вот и все! Результаты вы видите на рис. 7.6.

```
table {
    table-layout: fixed;
    width: 100%;
}
```

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**table-column-widths**



Благодарности

Спасибо **Крису Койеру** (Chris Coyier, http://css-tricks.com), придумавшему эту технику (http://css-tricks.com/fixing-tables-long-strings).

38 Стилизация путем подсчета смежных элементов

Проблема

Можно назвать много ситуаций, когда возникает необходимость определять для элементов разные стили в зависимости от того, сколько у них **всего** смежных элементов. Главный сценарий использования здесь — улучшение восприятия пользователем и экономия экранного пространства в раскрывающемся списке за счет скрывания элементов управления или уменьшения их в размере по мере увеличения списка. Вот несколько примеров:

- список сообщений электронной почты или схожих текстовых элементов. Если их всего лишь несколько штук, то для предварительного просмотра элементов мы можем отображать большие текстовые отрывки. По мере того как список растет, мы сокращаем количество строк в области предварительного просмотра. Когда длина списка становится больше высоты окна просмотра, мы можем даже полностью скрыть текстовые отрывки и уменьшить размер кнопок, для того чтобы минимизировать необходимость прокрутки;
- приложение с напоминаниями о предстоящих делах, в котором каждый элемент отображается крупным шрифтом, пока элементов не много. Чем больше элементов создает пользователь, тем меньше мы делаем размер шрифта (для всех элементов);
- приложение с цветовой палитрой, в котором элементы управления отображаются на каждом образце цвета. Эти элементы управления можно делать более компактными по мере того, как количество цветов растет, а место, занимаемое каждым образом, соответственно уменьшается (рис. 7.7);
- приложение с несколькими элементами `<textarea>`, в котором все их приходится уменьшать с добавлением каждого нового элемента (как на http://bytesizematters.com).

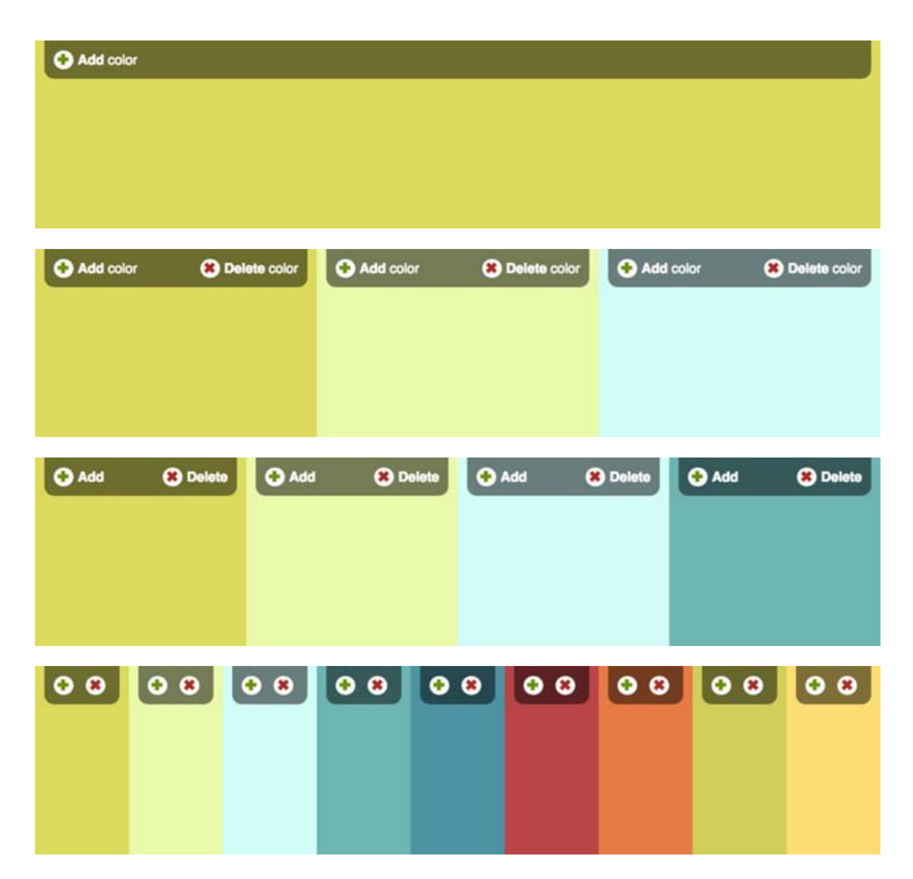


Рис. 7.7. Элементы управления становятся все меньше с увеличением количества цветов и сокращением количества свободного пространства. Обратите внимание на особое оформление в случае, когда цвет только один: кнопка удаления скрыта.
Цвета взяты из палитр Adobe Color (http://color.adobe.com):

Agave (http://color.adobe.com/agave-color-theme-387108)

Sushi Maki (http://color.adobe.com/Sushi-Maki-colortheme-350205)

Однако с помощью селекторов CSS решить задачу по выбору элементов в зависимости от общего количества «братьев» не так-то просто. Предположим, мы хотели бы применить определенные стили к элементам списка в случае, **когда общее количество элементов равно 4**. Мы могли бы использовать `li:nth-child(4)` для выбора четвертого элемента в списке, но это не то, что нам нужно; нам необходимо выбрать **все** элементы, но только в том случае, **когда их общее число равно четырем**.

Следующей идеей могло бы быть использование обобщенного комбинатора «братьев» (`~`) совместно с `:nth-child()`, как в `li:nth-child(4)`, `li:nthchild(4) ~ li`. Однако при этом в выборку попадают только четвертый потомок и элементы

после него (рис. 7.8), независимо от общего количества. Так как не существует комбинатора, который мог бы «оглянуться назад» и выбрать предыдущих «братьев», следует ли вовсе отказаться от попыток добиться нужного результата с помощью CSS? Нет, давайте не терять надежды.

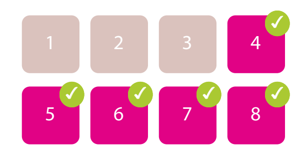


Рис. 7.8. Элементы, которые попадают в выборку при использовании li:nth-child(4), li:nthchild(4) ~ li

Решение

Для особого случая, когда элемент ровно **один**, существует очевидное решение: селектор `:only-child`, созданный специально для этого. Он не только удобен в качестве отправной точки; существуют несколько сценариев использования, требующих именно этого селектора, и поэтому он был добавлен в спецификацию. Например, обратите внимание, что на рис. 7.7 мы скрываем кнопку удаления, когда на экране остается только один цвет. Это можно реализовать с помощью селектора CSS `:only-child`:

```
li:only-child {
    /* Стили для ситуации, когда элемент только один */
}
```

Но селектор `:only-child` эквивалентен `:first-child:last-child`. Причина очевидна: если **первый** элемент также является **последним** элементом, то отсюда логически вытекает, что он **единственный** элемент. Кроме того, `:last-child` — это сокращение для `:nthlast-child(1)`:

```
li:first-child:nth-last-child(1) {
    /* То же самое, что и li:only-child */
}
```

Теперь у нас есть параметр — в данном случае это 1, — и мы можем настраивать его по своему усмотрению. Попробуйте угадать, какие элементы выбирает селектор `li:first-child:nth-last-child(4)`. Если ваш ответ заключается в том, что это обобщение `:only-child` для выбора элементов списка, когда их общее количество равно четырем, то вы слишком оптимистичны. Мы еще не достигли желаемой цели, хотя уже находимся на правильном пути. Попробуйте думать об этих псевдоклассах по отдельности: мы ищем элементы, которые подходят под **оба** условия: **и** `:first-child`, **и** `:nth-last-child(4)`. Следовательно, нас интересуют элементы, одновременно являющиеся первым потомком своего родителя, если считать от начала, и четвертым потомком, если считать от конца. Какие элементы удовлетворяют этим критериям?

Ответ прост: **первый элемент в списке, состоящем в точности из четырех элементов** (рис. 7.9). Это не совсем то, чего мы хотели, но очень близко: поскольку

В этом разделе мы будем использовать селекторы `:nth-child()`, но обсуждение в равной степени применимо и к селекторам `:nth-of-type()`, которые нередко оказываются лучшим выбором, так как чаще всего братья принадлежат к разным типам, а нас интересует только один. В примерах мы будем работать с элементами списка, но обсуждаемые методы также можно использовать и с элементами любого другого типа.

теперь мы знаем, как выделить первого потомка в таком списке, мы можем использовать общий комбинатор потомков (~) для выбора **каждого потомка, следующего** за таким первым потомком. По сути, мы выберем **все элементы в списке в том и только том случае, когда список состоит из четырех элементов**, а это как раз то, чего мы пытаемся добиться:

```
li:first-child:nth-last-child(4),
li:first-child:nth-last-child(4) ~ li {
    /* Выбор всех элементов списка, если список
       содержит ровно четыре элемента */
}
```

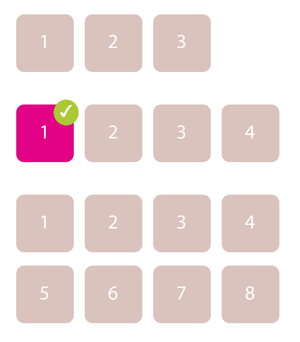


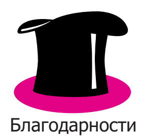
Рис. 7.9. Какие элементы попадают в выборку при использовании li:first-child:nth-last-child(4) на списке из трех, четырех и восьми элементов

Для того чтобы избежать разрастания кода и повторений в продемонстрированном выше решении, можно прибегнуть к помощи препроцессора, такого как SCSS, хотя синтаксис существующих препроцессоров для подобных вещей довольно неповоротлив:

SCSS

```
/* Определение примеси */
@mixin n-items($n) {
    &:first-child:nth-last-child(#{$n}),
    &:first-child:nth-last-child(#{$n}) ~ & {
        @content;
    }
}

/* Использовать так: */
li {
    @include n-items(4) {
        /* Свойства и значения */
    }
}
```



Спасибо **Андрэ Луису** (André Luís, http://andr3.net) за идею, послужившую вдохновением для данной техники (http://andr3.net/blog/post/142).

Выбор по диапазону количества смежных элементов

В самых практичных приложениях мы хотели бы задавать не конкретные количества элементов, а некие диапазоны. Есть удобный трюк, позволяющий заставлять селекторы `:nth-child()` выбирать диапазоны, например: *«все после четвертого потомка»*. Помимо обычных чисел, в качестве параметров для них можно также указывать выражения в формате `an+b` (скажем, `:nth-child(2n+1)`), где `n` — переменная в диапазоне от 0 до +∞ в теории (на практике значения после определенного лимита перестают выбирать что-либо, так как количество элементов у нас конечно). Если мы используем выражение в форме `n+b` (подразумевается, что `a` равно 1), то не существует положительного целого `n`, которое могло бы дать нам значение, меньшее `b`. Следовательно, выражения в форме `n+b` можно использовать для выбора **всех потомков, начиная с `b`-го и дальше**. Например, `:nth-child(n+4)` выбирает всех потомков, за исключением первого, второго и третьего (рис. 7.10).

СОВЕТ

Разобраться, как правильно применять селекторы `:nth-*`, может быть невероятно сложно. Если вы столкнулись с проблемами, то можете воспользоваться интерактивным тестовым приложением, позволяющим поэкспериментировать с несколькими выражениями. Написанное мной вы найдете на странице http://lea.verou.me/demos/nth.html, но в Сети также существует множество других.

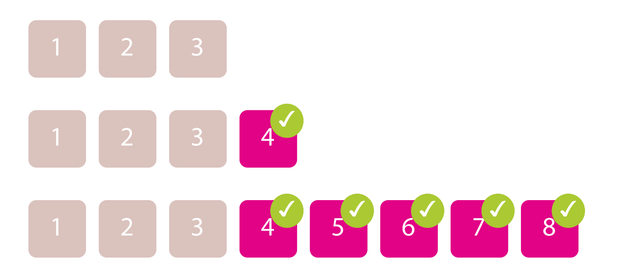


Рис. 7.10. Какие элементы попадают в выборку при использовании li:nth-child(n+4) на списке из трех, четырех и восьми элементов

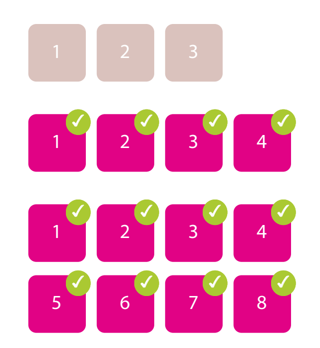


Рис. 7.11. Какие элементы попадают в выборку при использовании li:first-child:nth-last-child(n+4), li:first-child:nth-last-child(n+4) ~ li на списке из трех, четырех и восьми элементов

Мы можем использовать это полезное качество для выбора элементов списка в ситуации, когда общее количество элементов составляет четыре или больше (рис. 7.11). В данном случае в качестве выражения, передаваемого `:nth-last-child()`, следует использовать `n+4`:

```
li:first-child:nth-last-child(n+4),
li:first-child:nth-last-child(n+4) ~ li {
	/* Выбор всех элементов списка, если список содержит
	   по меньшей мере четыре элемента */
}
```

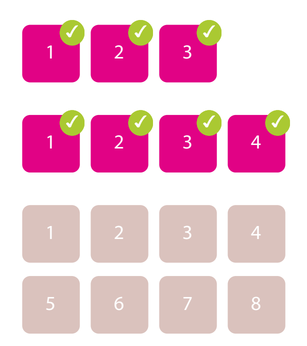


Рис. 7.12. Какие элементы попадают в выборку при использовании li:first-child:nth-last-child(-n+4), li:first-child:nth-last-child(-n+4) ~ li на списке из трех, четырех и восьми элементов

Схожим образом, выражения в форме `-n+b` можно использовать для выбора **первых `b` элементов**. То есть для выбора всех элементов списка в том и только том случае, если общее количество элементов в одном и том же списке равно **четырем или менее**, мы бы написали:

```
li:first-child:nth-last-child(-n+4),
li:first-child:nth-last-child(-n+4) ~ li {
    /* Выбор всех элементов списка, если список
           содержит максимум четыре элемента */
}
```

Разумеется, мы бы могли объединить два решения, но код при этом станет еще более тяжеловесным. Предположим, что мы хотим выбрать все элементы списка, когда в списке содержится от 2 до 6 элементов:

```
li:first-child:nth-last-child(n+2):nth-last-child(-n+6),
li:first-child:nth-last-child(n+2):nth-last-child(-n+6) ~ li {
    /* Выбор всех элементов списка, если список содержит
       от 2 до 6 элементов */
}
```

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**styling-sibling-count**

СВЯЗАННЫЕ СПЕЦИФИКАЦИИ

Selectors: http://w3.org/TR/selectors

39 Текучий фон, фиксированное содержимое

Проблема

В последние несколько лет в веб-дизайне набирает популярность новое направление, которое я называю *«текучая ширина фона, фиксированная ширина содержимого»*. Типичные характеристики подобного шаблона включают:

- несколько разделов, каждый из которых занимает всю ширину просмотрового окна, с разным оформлением фона;
- ширина содержимого фиксирована, даже если конкретное значение ширины варьируется в зависимости от разрешения экрана, так как для управления шириной используются медиазапросы. В некоторых случаях ширина содержимого в разных разделах может различаться.

Иногда веб-сайт целиком состоит из разделов, стилизованных вышеописанным образом (рис. 7.15 или более изящный вариант на рис. 7.14). Чаще всего данный вариант оформления используется только для отдельных разделов, в частности нижнего колонтитула (рис. 7.13).



Рис. 7.13. На веб-сайте популярного сервиса аренды квартир http://airbnb.com этот шаблон применяется в нижнем колонтитуле

Самый распространенный вариант реализации подобного дизайна заключается в использовании **двух элементов для каждого раздела**: один для текучего фона, а второй для содержимого фиксированной ширины. Второй элемент выравнивается по центру с помощью `margin: auto`. Например, разметка для такого нижнего колонтитула может выглядеть приблизительно так:

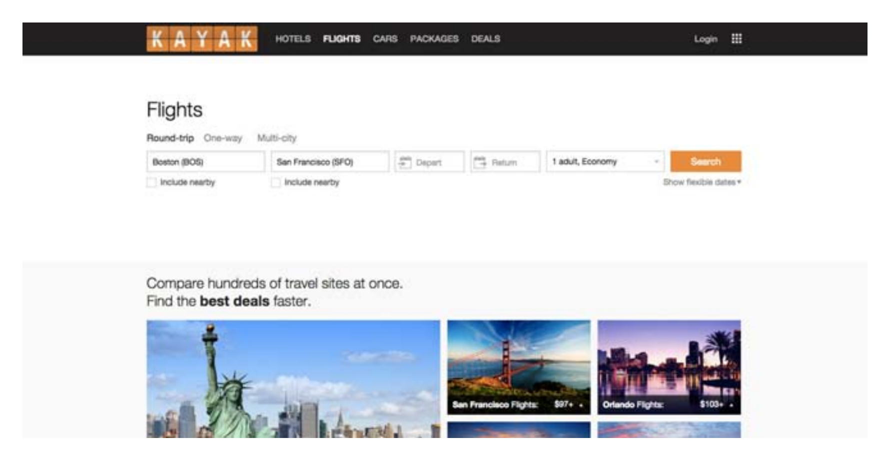


Рис. 7.14. На популярном веб-сайте для путешественников http://kayak.com данный шаблон используется для оформления всей домашней страницы, но очень изящным ненавязчивым способом

HTML

```
<footer>
    <div class="wrapper">
        <!-- Здесь находится содержимое нижнего колонтитула -->
    </div>
</footer>
```

CSS-код обычно включает правила, структурированные как в примере ниже:

```
footer {
    background: #333;
}
.wrapper {
    max-width: 900px;
    margin: 1em auto;
}
```

Выглядит знакомо? Большинству веб-дизайнеров и разработчиков приходилось на том или ином этапе своей карьеры писать подобный код. Действительно ли дополнительные элементы — это неизбежное зло или современный уровень CSS позволяет избежать их использования?

> **!** Не забывайте в функции `calc()` вокруг всех операторов `-` и `+` добавлять пробелы, иначе ваш код вернет ошибку при разборе! Причина этого странного правила кроется в обеспечении совместимости в будущем: возможно, позднее в `calc()` будут разрешены идентификаторы, а они могут содержать дефисы.

Решение

Давайте подумаем, какую роль в данном случае играет `margin: auto`. Поле, которое создается этим правилом, равно половине ширины просмотрового

окна за вычетом половины ширины страницы. Так как процентные значения у нас завязаны на ширину окна просмотра (при условии, что у него не существует предка с явно определенной шириной), в нашем случае это выглядит как `50% - 450px`. Однако функция `calc()`, определенная в **CSS Values and Units Level 3** (http://w3.org/TR/css-values-3/#calc), позволяет использовать такие простые математические выражения прямо в таблице стилей. Заменив `auto` на `calc()`, мы получим такое правило-обертку:

```
.wrapper {
    max-width: 900px;
    margin: 1em calc(50% - 450px);
}
```

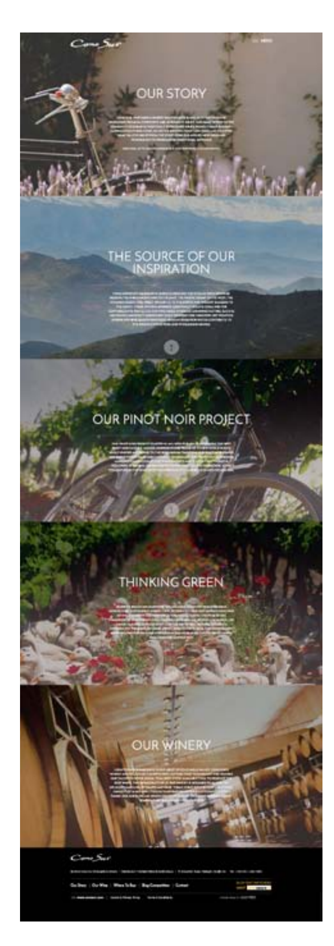


Рис. 7.15. Превосходный веб-сайт ирландской винодельни Cono Sur Vineyards and Winery (http://conosur.ie) — пример интенсивного использования этого шаблона

Единственная причина, почему нам требовался второй элемент-обертка, — необходимость применять волшебное ключевое слово `auto` к его полю посредством свойства `margin`. Однако мы избавились от колдовства и заменили его обычным `calc()`, то есть теперь это всего лишь еще один размер в CSS, который можно указывать для любого свойства, принимающего подобные значения. Таким образом, при желании мы можем использовать его с родительским элементом в свойстве `padding`:

```
footer {
    max-width: 900px;
    padding: 1em calc(50% - 450px);
    background: #333;
}
.wrapper {}
```

Как вы видите, благодаря этому мы убрали весь CSS-код из обертки, что означает, что она нам больше не требуется и мы можем спокойно удалить ее из нашей разметки. Мы создали требуемый стиль безо всяких лишних элементов HTML. Можно ли дополнительно улучшить его? Как всегда, ответ положительный.



При использовании этого решения забивка может вовсе исчезнуть, если ширина экрана окажется меньше ширины содержимого. Но это можно поправить с помощью медиазапросов.

Обратите внимание, что если закомментировать объявление `width`, то ничего не произойдет.

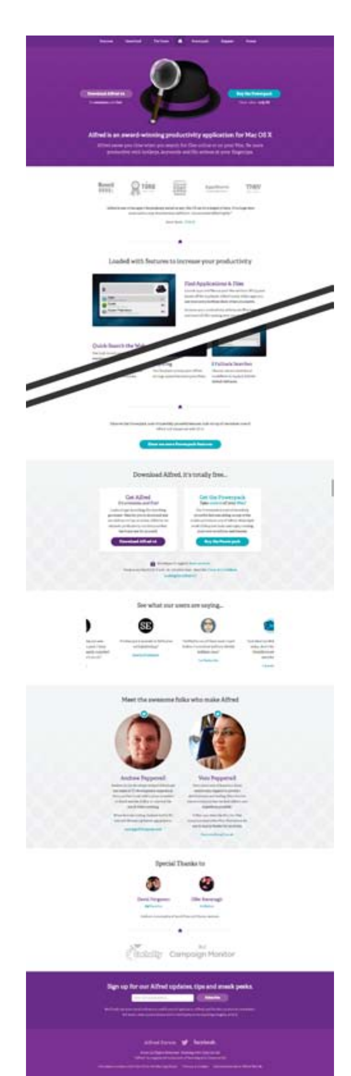


Рис. 7.16. На веб-сайте Alfred (https://www.alfredapp.com), популярного приложения для Mac OS, предназначенного для повышения производительности работы с компьютером, также используется этот стиль

Визуальный результат будет точно таким же, и поведение тоже не изменится, независимо от размера окна просмотра. Почему же? Потому что забивка, равная `50% - 450px`, все равно оставляет только `900px` (`2 × 450px`) доступного пространства. Мы бы увидели отличия, если бы значение `width` отличалось от `900px` в большую или меньшую сторону. Но мы в любом случае получаем именно `900px`, поэтому объявление ширины избыточно, и мы можем убрать его, следуя заветам DRY.

Еще одно усовершенствование не помешало бы, чтобы обеспечить обратную совместимость. Нам следует добавить резервное решение, чтобы получать **хоть какую-то** забивку в ситуации, когда `calc()` не поддерживается:

```
footer {
    padding: 1em;
    padding: 1em calc(50% - 450px);
    background: #333;
}
```

Готово! Мы добились гибкого, соответствующего принципам DRY и обеспечивающего обратную совместимость результата, написав всего лишь три строки CSS-кода безо всякой лишней разметки!

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**fluid-fixed**

СВЯЗАННЫЕ СПЕЦИФИКАЦИИ

CSS Values & Units: http://w3.org/TR/css-values

40 Центрирование по вертикали

Проблема

44 года назад мы высадили человека на Луне, но до сих пор не можем центрировать объекты по вертикали в CSS.

— **Джеймс Андерсон (James Anderson**,
http://twitter.com/jsa/status/358603820516917249)

Центрировать элементы **по горизонтали** в CSS невероятно просто: если это строковый элемент, то мы применяем `text-align: center` к его предку, а если это блочный элемент, то мы применяем `margin: auto` к нему самому. Но одной мысли о том, чтобы центрировать элемент **по вертикали**, достаточно, чтобы по спине пробежали холодные мурашки.

С годами центрирование по вертикали превратилось в «святой грааль» CSS и «семейную шутку» разработчиков внешних интерфейсов. И это понятно, ведь здесь налицо все необходимые причины:

- это то, что требуется разработчикам очень часто;
- это звучит чрезвычайно просто в теории;
- на практике это было невероятно сложно, особенно для элементов с переменными габаритными размерами.

Разработчики внешних интерфейсов уже исчерпали свою фантазию в попытках придумать новые выходы из этого тупика, и большинство придуманных ими решений ужасающе грязные. В этом секрете мы исследуем некоторые из лучших современных техник, позволяющих реализовать вертикальное центрирование в любых ситуациях. Обратите внимание, что несколько существующих популярных техник здесь не упоминаются по разным причинам:

- **метод с табличным макетом** (использующий режимы отображения таблиц) не включен, так как требует нескольких лишних элементов HTML;
- **метод со строковым блоком** не включен, так как, на мой вкус, он слишком грязный.

Однако если вам интересно, вы можете прочитать об обеих этих техниках в великолепной статье Криса Койера *Centering in the Unknown* (http://csstricks.com/centering-in-the-unknown).

Если не указано иное, мы будем использовать следующую разметку прямо внутри элемента `<body>`, хотя решения, которые мы собираемся изучить, должны работать вне зависимости от выбранного вами контейнера:

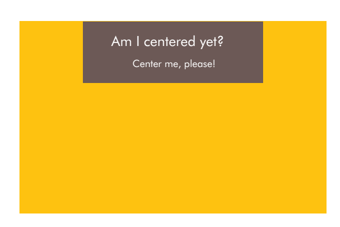


Рис. 7.17. Наша отправная точка

HTML

```
<main>
    <h1>Am I centered yet?</h1>
    <p>Center me, please!</p>
</main>
```

Мы также применим несколько простейших правил CSS для оформления фона, забивки и т. п., чтобы наша отправная точка выглядела как на рис. 7.17.

Решение с абсолютным позиционированием

Одна из ранних техник вертикального центрирования требовала фиксированных значений ширины и высоты:

```
main {
    position: absolute;
    top: 50%;
    left: 50%;
    margin-top: -3em; /* 6/2 = 3 */
    margin-left: -9em; /* 18/2 = 9 */
    width: 18em;
    height: 6em;
}
```

По сути, здесь мы помещаем верхний левый угол элемента в центр окна просмотра (или ближайшего по расположению предка), а благодаря отрицательным значениям полей, равным половине ширины и высоты элемента, перемещаем его вверх и влево, так, чтобы **центр элемента совпал с центром просмотрового окна**. С помощью `calc()` это решение можно сделать на два объявления проще:

```
main {
    position: absolute;
    top: calc(50% - 3em);
    left: calc(50% - 9em);
    width: 18em;
    height: 6em;
}
```

Очевидно, что самая большая проблема этой техники в том, что она зависит от фиксированных габаритных размеров, тогда как нам часто требуется центрировать элементы, размеры которых определяются их содержимым. Если бы только мы могли использовать процентные значения, разрешающиеся в габаритные размеры элемента, наша проблема была бы решена. К сожалению, для большинства свойств CSS (включая `margin`) процентные значения разрешаются относительно габаритных размеров родительского элемента.

Как это часто бывает с CSS, решения приходят из самых неожиданных мест. В данном случае нам могут помочь трансформации CSS. Когда мы используем процентные значения в трансформациях `translate()`, мы перемещаем элементы относительно их собственных ширины и высоты — а это в точности то, что нам нужно! Таким образом, мы можем заменить отрицательные смещения, жестко кодирующие габаритные размеры наших элементов, трансформациями CSS, основанными на процентных значениях. Это позволит нам избавиться от жестко закодированных значений:

```
main {
    position: absolute;
    top: 50%;
    left: 50%;
    transform: translate(-50%, -50%);
}
```

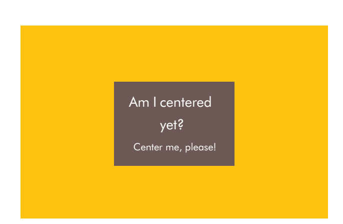


Рис. 7.18. Вертикальное центрирование без указания конкретных габаритных значений благодаря нашему трюку с трансформациями CSS

Результат вы можете видеть на рис. 7.18, но там нет ничего удивительного: контейнер идеально выровнен по центру, как и ожидалось.

Разумеется, ни одна техника не идеальна, и у этой также есть несколько недостатков:

- во многих ситуациях абсолютное позиционирование применять невозможно, так как оно способно радикально изменить весь макет целиком;
- если центрируемый элемент выше своего просмотрового окна, то он обрезается сверху



Рис. 7.19. Если элемент, который мы пытаемся центрировать по вертикали, выше своего окна просмотра, то он обрезается сверху

(рис. 7.19). Справиться с этим можно несколькими способами, но все они невероятно грязные;

- в некоторых браузерах это может привести к тому, что элементы выглядят слегка нечеткими, так как позиционируются с точностью до половины пиксела. Это можно исправить, применив `transform-style: preserve-3d`, но это грязный трюк, и невозможно гарантировать, что он продолжит работать в будущем.

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**vertical-centering-abs**



Благодарности

Как выяснилось, очень трудно отследить, кто первым додумался до этого полезного трюка, но самым ранним источником кажется пользователь веб-сайта StackOverflow (http://stackoverflow.com) с псевдонимом **Charlie** (http://stackoverflow.com/users/479836/charlie), который опубликовал описание в ответе на вопрос Align vertically using CSS 3? (http://stackoverflow.com/a/16026893/90826) 16 апреля 2013 года.

Решение с единицами измерения просмотрового окна

Даже если бы мы хотели избежать абсолютного позиционирования, мы все так же можем использовать трюк с трансформацией `translate()`, для того чтобы перемещать элемент на половину его ширины и высоты. Но как в этом случае задать изначальное смещение на 50% от верхнего левого угла контейнера, не используя `left` и `top`?

Первой мыслью могло бы быть использование процентных значений со свойством `margin`, например так:

```
main {
    width: 18em;
    padding: 1em 1.5em;
    margin: 50% auto 0;
    transform: translateY(-50%);
}
```

Однако, как вы видите на рис. 7.20, это дает несколько странный результат. Причина в том, что **процентные значения в свойстве `margin` вычисляются относительно ширины его родительского элемента**. Да, даже проценты для `margin-top` и `margin-bottom`!

К счастью, если мы пытаемся центрировать элемент относительно окна просмотра, у нас еще есть надежда. В спецификации **CSS Values and Units Level 3** (http://w3.org/TR/css-values-3/#viewport-relative-lengths) определяется семейство новых единиц измерения — значений, завязанных на размер просмотрового окна:

- `vw` относится к **ширине просмотрового окна**. Вопреки ожиданиям, `1vw` обозначает 1% ширины просмотрового окна, а не 100%;
- аналогично `vw`, `1vh` представляет 1% **высоты просмотрового окна**;
- `1vmin` эквивалентно `1vw`, если ширина просмотрового окна меньше его высоты; в противном случае это значение эквивалентно `1vh`;
- `1vmax` эквивалентно `1vw`, если ширина просмотрового окна больше его высоты; в противном случае это значение эквивалентно `1vh`.

В нашей ситуации для задания ширины полей нам требуется `vh`:

```
main {
    width: 18em;
    padding: 1em 1.5em;
    margin: 50vh auto 0;
    transform: translateY(-50%);
}
```

Как подтверждает рис. 7.21, это решение работает безупречно. Разумеется, возможности применения данной техники сильно ограниченны, так как единственная задача, решаемая здесь, — это центрирование по вертикали относительно окна просмотра.

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**vertical-centering-vh**

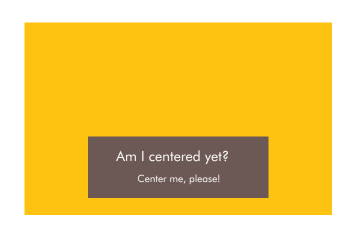


Рис. 7.20. Попытка сослаться на габаритные размеры просмотрового окна с помощью процентных значений в свойстве margin не приводит к желаемому результату

Обратите внимание, что, используя значения, определяемые относительно размеров просмотрового окна, можно также создавать полноэкранные разделы, совершенно не прибегая к помощи сценариев. Более подробное описание вы найдете в статье Эндрю Скора *Make full screen sections with 1 line of CSS* (Andrew Ckor, http://medium.com/@ckor/make-full-screen-sections-with-1-line-of-css-b82227c75cbd).

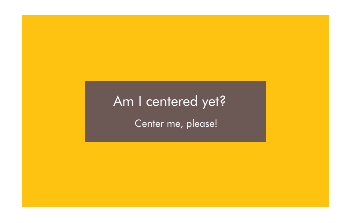


Рис. 7.21. Использование 50vh в качестве размера верхнего поля решает нашу проблему, и теперь элемент успешно центрируется по вертикали

Решение с гибким полем

Это, несомненно, наилучшее из доступных решений, так как гибкое поле (спецификация **Flexbox**, http://w3.org/TR/css-flexbox) разработано специально для помощи в подобных ситуациях. Единственная причина, почему мы все еще рассматриваем другие решения, заключается в том, что прочие методы лучше поддерживаются браузерами, хотя и у гибкого поля достаточно хорошая поддержка современными браузерами.

Все, что нам требуется, — это два объявления: `display: flex` на родительском элементе относительно центрируемого (в нашем примере это элемент `<body>`) и уже знакомый нам `margin: auto` на дочернем, который мы центрируем (в нашем примере это `<main>`):

```
body {
    display: flex;
    min-height: 100vh;
    margin: 0;
}

main {
    margin: auto;
}
```

Обратите внимание, что при использовании гибкого поля `margin: auto` выравнивает элемент по центру не только по горизонтали, но и по вертикали. Также обратите внимание, что нам даже не пришлось задавать ширину (хотя при желании мы могли бы это сделать): присвоенная ширина эквивалентна `max-content`. (Помните ключевые слова для определения размера изнутри, о которых я рассказывала в **секрете «Определение размера изнутри»**?)

Если гибкое поле не поддерживается, то результат выглядит в точности так же, как наша отправная точка на рис. 7.17 (если задано конкретное значение ширины), что вполне допустимо, даже если желаемого вертикального центрирования мы не добились.

Еще одно преимущество гибкого поля заключается в том, что с помощью него можно вертикально центрировать анонимные контейнеры (то есть текст безо всякой обертки). Например, если бы мы использовали такую разметку:

HTML

```
<main>Center me, please!</main>
```

то могли бы задать фиксированные значения для `main` и **выровнять текст по центру прямо внутри этого тега**, используя свойства `align-items` и `justify-content`, которые были добавлены в спецификации Flexbox (рис. 7.22):

```
main {
    display: flex;
    align-items: center;
    justify-content: center;
    width: 18em;
    height: 10em;
}
```

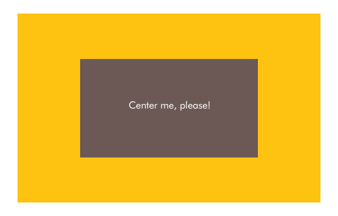


Рис. 7.22. Использование гибкого поля для центрирования анонимных текстовых полей

Те же свойства можно было бы использовать и с `<body>` для центрирования элемента `<main>`, но подход с `margin: auto` элегантнее, а также обеспечивает обходной путь.

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**vertical-centering**

СВЯЗАННЫЕ СПЕЦИФИКАЦИИ

CSS Transforms: http://w3.org/TR/css-transforms

CSS Values & Units: http://w3.org/TR/css-values

CSS Flexible Box Layout: http://w3.org/TR/css-flexbox

CSS Box Alignment: http://w3.org/TR/css-align

БУДУЩЕЕ.
ВЫРОВНЯТЬ ВСЕ!

Как уже запланировано в спецификации CSS Box Alignment Level 3 (http://w3.org/TR/css-align-3), в будущем нам не придется использовать другой режим разметки для обеспечения возможности вертикального центрирования. Мы сможем делать это, используя простую строку кода:

```
align-self: center;
```

Она будет работать независимо от того, какие другие свойства определены для элемента. Кажется, что это слишком хорошо, для того чтобы быть правдой, но совсем скоро вы сможете использовать это решение в любимом браузере!

41 Липкие нижние колонтитулы

ПРЕДВАРИТЕЛЬНЫЕ ТРЕБОВАНИЯ

Единицы измерения, завязанные на окно просмотра (см. секрет «Центрирование по вертикали»), функция `calc()`

В частности, эта проблема возникает на страницах, содержимое которых короче высоты просмотрового окна за вычетом высоты нижнего колонтитула.

Проблема

Это одна из старейших и самых известных проблем веб-дизайна. Она встречается настолько часто, что большинство из нас обязательно сталкивались с ней в тот или иной момент своей карьеры. В двух словах смысл ее таков: нижний колонтитул с любой блочной стилизацией, такой как фон или тень, прекрасно работает, когда содержимое достаточно длинное, но ломается на коротких страницах (таких, как сообщения об ошибке). Поломка выглядит так, что нижний колонтитул «прилипает» не к нижней кромке окна просмотра, как нам хотелось бы, а к нижней кромке содержимого.

Объясняется такая популярность этой проблемы не только ее вездесущностью, но также тем, насколько **обманчиво простой она кажется с первого взгляда**. Это классический случай задачи, на решение которой приходится затратить намного больше времени, чем поначалу ожидалось. Помимо этого, **средствами CSS 2.1 она все еще не решается**: практически все классические решения требуют указывать фиксированную высоту нижнего колонтитула, что надуманно, а зачастую попросту невозможно. Кроме того, все эти решения **чрезмерно сложные, грязные** и предъявляют **специфические требования к разметке**. Но тогда ничего лучше в нашем распоряжении не было, учитывая ограничения

CSS 2.1. Однако с современным CSS мы можем намного больше! Так как же решить задачу?

Решение с фиксированной высотой

Мы будем работать с очень простой страницей, внутри элемента `<body>` которой содержится следующая разметка:

HTML

```
<header>
    <h1>Site name</h1>
</header>
<main>
    <p>Bacon Ipsum dolor sit amet…
    <!-- Текстовая забивка с веб-сайта baconipsum.com --></p>
</main>
<footer>
    <p>© 2015 No rights reserved.</p>
    <p>Made with ♥ by an anonymous
                 pastafarian.</p>
</footer>
```

Мы также определили несколько простых стилей для нашей страницы, в том числе добавили фон к нижнему колонтитулу. Ее текущий вид представлен на рис. 7.23. Теперь давайте немного уменьшим объем содержимого. Результат этой операции вы видите на рис. 7.24. Проблема липкого нижнего колонтитула во всей красе! Великолепно, мы воспроизвели проблему, но как нам ее решить?

Предположим, что текст в нижнем колонтитуле никогда не будет переноситься на новую строку. Тогда мы можем вычислить его высоту в формате, подходящем для использования в CSS-коде:

$$2 \text{ строки} \times \text{высота строки} + 3 \times \text{поле абзаца} + \\ + \text{забивка по вертикали} = \\ = 2 \times 1.5\text{em} + 3 \times 1\text{em} + 1\text{em} = 7\text{em}$$

Аналогично, высота заголовка равна 2.5em. Следовательно, используя единицы измерения, привязанные к окну просмотра, и функцию `calc()`, мы можем «прилепить» наш нижний колонтитул

Если вам никогда не приходилось рвать на себе волосы, погружаясь в дебри существующей литературы на эту тему, то вот несколько популярных ссылок, где вы найдете описание часто используемых решений, не раз выручавших веб-разработчиков до того, как спецификации CSS Level 3 появились хотя бы в проекте:

http://cssstickyfooter.com

http://ryanfait.com/sticky-footer

http://css-tricks.com/snippets/css/sticky-footer

http://pixelsvsbytes.com/blog/2011/09/sticky-css-footersthe-flexible-way

http://mystrd.at/modern-clean-csssticky-footer

Последние два упрощены до невозможности, но все же накладывают собственные ограничения использования.

к нижней кромке, используя, по сути, одну строку CSS-кода:

```
main {
    min-height: calc(100vh - 2.5em - 7em);
    /* Нужно избежать забивки/рамок и прочих
            игр с высотой: */
    box-sizing: border-box;
}
```

Рис. 7.23. Как наша простая страница выглядит, когда содержимое достаточно длинное

В качестве альтернативы мы могли бы создать обертку вокруг наших элементов `<header>` и `<main>`, для того чтобы оставалось только вычислить высоту нижнего колонтитула:

```
#wrapper {
    min-height: calc(100vh - 7em);
}
```

Рис. 7.24. Проблема липкого нижнего колонтитула во всем великолепии

Это работает (рис. 7.25), и присущий данному решению минимализм делает его несколько лучше существующих решений с фиксированной высотой. Однако оно совершенно непрактично, и его невозможно использовать нигде, кроме веб-сайтов с самыми простыми вариантами разметки. Оно основано на предположении, что в нижнем колонтитуле не будет переносов строки, значение `min-height` необходимо корректировать каждый раз, когда меняются размеры нижнего колонтитула (то есть о принципах DRY можно забыть), и если только мы не согласны обернуть наш заголовок и содержимое в еще один элемент HTML, те же вычисления и модификации необходимо делать также и для заголовка. Определенно, в наши дни должен существовать лучший путь!

Рис. 7.25. Нижний колонтитул после того, как мы приклеили его к нужному месту с помощью CSS

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**sticky-footer-fixed**

Гибкое решение



Соблюдайте осторожность, используя сложение и вычитание в функции `calc()`: вокруг операторов `+` и `-` **обязательно** нужно добавлять пробелы. Это странное решение было принято для обеспечения совместимости в будущем. Если в какой-то момент в `calc()` разрешат использовать ключевые слова, синтаксическому анализатору CSS нужно будет как-то отличать дефис в ключевом слове от оператора вычитания.

Гибкое поле идеально подходит для решения подобных задач. Мы можем достичь превосходной гибкости с помощью всего нескольких строк CSS-кода, и нам не потребуются корявые вычисления и дополнительные HTML-элементы. Во-первых, мы должны применить `display: flex` к элементу `<body>`, так как это родительский элемент всех трех наших главных блоков. Это включит разметку гибкого поля для всех них. Также нам нужно для свойства `flex-flow` установить значение `column`, иначе блоки будут выводиться друг за другом по горизонтали в одной строке (рис. 7.26):

```
body {
    display: flex;
    flex-flow: column;
}
```

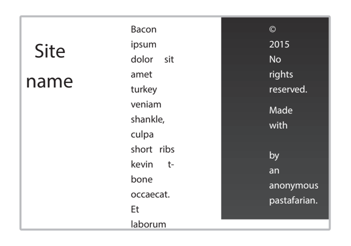


Рис. 7.26. Использование flex без применения других свойств выстраивает дочерние элементы нашего элемента в одну строку

Пока наша страница выглядит практически так же, как до применения всех этих хитростей гибкого поля, так как каждый элемент растянут на всю ширину просмотрового окна, а его размер определяется его содержимым. Ну конечно, ведь мы же еще не воспользовались преимуществом гибкого поля!

Для того чтобы чудо произошло, нам нужно задать значение `100vh` для свойства `min-height` элемента `<body>`, чтобы он занимал **по меньшей мере всю высоту окна просмотра**. Пока что результат все так же выглядит, как на рис. 7.24, поскольку мы, конечно, указали минимальную высоту для всего элемента `<body>`, но высота каждого поля все так же определяется его содержимым (то есть *определяется изнутри*, если говорить на профессиональном жаргоне CSS).

Мы должны сделать так, чтобы высота заголовка и нижнего колонтитула определялась **изнутри** но при этом высота содержимого гибко менялась, растягиваясь на все оставшееся пространство. Этого можно добиться, задав большее нуля (подойдет `1`) значение `flex` для контейнера `<main>`:

СОВЕТ

Свойство `flex` — это сокращение, объединяющее свойства `flex-grow`, `flex-shrink` и `flex-basis`. Любой элемент, для которого определено значение `flex` больше `0`, становится гибким, а кроме того, `flex` управляет отношением между габаритными размерами различных гибких элементов. Например, в нашем случае если бы для `<main>` мы определили `flex: 2`, а для `<footer>` — `flex: 1`, то высота содержимого в **два** раза превышала бы высоту нижнего колонтитула. То же самое произошло бы с парой значений `4` и `2` вместо `2` и `1`, так как важны не абсолютные значения, а отношение между ними.

```
body {
    display: flex;
    flex-flow: column;
    min-height: 100vh;
}

main { flex: 1; }
```

Вот и всё, больше никакой код не требуется! Идеально липкий нижний колонтитул (визуально наш результат выглядит так же, как на рис. 7.25) — и всего лишь четыре простые строки кода! Ну разве не чудо это гибкое поле?

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**sticky-footer**


Благодарности

Спасибо Филипу Уолтону (Philip Walton, http://philipwalton.com) за изобретение этой техники (http://philipwalton.github.io/solved-by-flexbox/demos/sticky-footer).

СВЯЗАННЫЕ СПЕЦИФИКАЦИИ

CSS Flexible Box Layout: http://w3.org/TR/css-flexbox

CSS Values & Units: http://w3.org/TR/css-values

Переходы и анимация



42 Эластичные переходы

ПРЕДВАРИТЕЛЬНЫЕ ТРЕБОВАНИЯ

Базовые знания о переходах CSS, базовые знания об анимации CSS

Почему мы используем трансформации, а не какое-нибудь другое свойство CSS, такое как `top` или `margin-top`? На момент написания этой главы трансформации создают намного более плавные эффекты, тогда как прочие свойства CSS чаще всего заставляют элемент прилипать к пиксельным границам.

Проблема

Эластичные переходы и анимация (то есть пружинящие переходы) — это популярный способ придания интерфейсу жизнерадостности и реалистичности, ведь когда объекты движутся в реальной жизни, они редко перемещаются из точки А в точку Б, не демонстрируя никакой упругости.

С технической точки зрения пружинящий эффект заключается в том, что после того, как переход достигает конечного значения, он немного отматывается назад, затем снова достигает конечного значения, снова отматывается назад, но уже на меньшее значение, и так повторяется один или несколько раз до тех пор, пока переход окончательно не завершится. Например, предположим, что мы анимируем элемент, стилизованный под падающий мяч (рис. 8.1), определяя с помощью `transform` переход от `none` до `translateY(350px)`.

Разумеется, элемент может демонстрировать пружинящее поведение не только при перемещении по плоскости. Этот эффект способен сделать значительно привлекательнее любой тип перехода, включая:

- переходы, затрагивающие размер (например, можно увеличивать элемент при срабатывании `:hover`, отображать растущее в размере диалоговое окно, начиная с `transform: scale(0)`, анимировать столбики на диаграмме);

- переходы, включающие изменение угла (например, вращения или секторные диаграммы, секторы на которых посредством анимации вырастают с нуля).

Несколько библиотек JavaScript содержат встроенные возможности по созданию пружинящей анимации. Однако сегодня мы более не нуждаемся в помощи сценариев для добавления в наши проекты анимации и переходов. Как же тогда наилучшим способом реализовать пружинящий эффект средствами CSS?

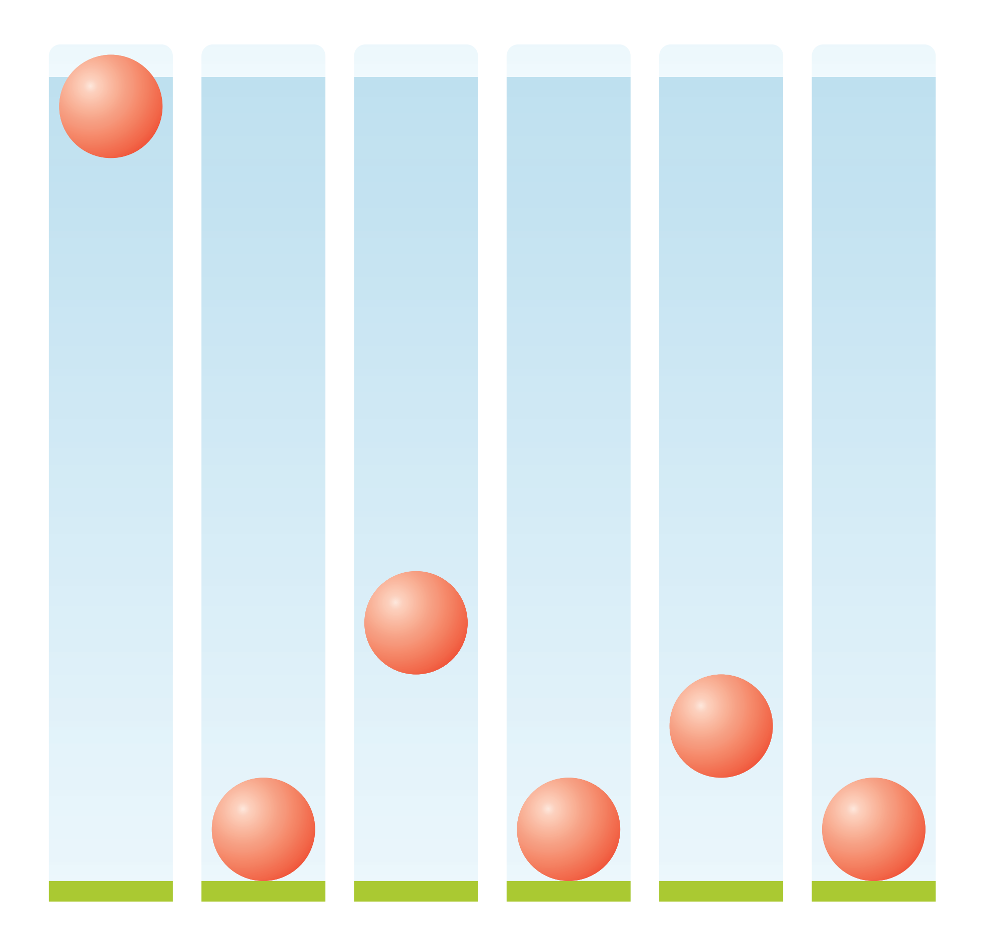


Рис. 8.1. Пружинящее движение в реальной жизни

Пружинящая анимация

Нашей первой идеей может быть использование анимации CSS с ключевыми кадрами, как в следующем фрагменте кода:

```
@keyframes bounce {
    60%, 80%, to { transform: translateY(350px); }
    70% { transform: translateY(250px); }
    90% { transform: translateY(300px); }
}

.ball {
    /* Размеры, цвета и т. п. */
    animation: bounce 3s;
}
```

Ключевые кадры в этом фрагменте кода соответствуют шагам, представленным на рис. 8.1. Однако если вы запустите эту анимацию, то заметите, что она выглядит искусственной. Одна из причин, почему так происходит, — каждый раз, когда мячик меняет направление, он продолжает ускоряться, что выглядит очень неестественно. Корень зла здесь в том, что *функция расчета времени* для этого объекта одна и та же во всех ключевых кадрах.

«Функция... чего?» — спросите вы. С каждым переходом и анимацией связана **кривая, определяющая, как этот эффект развивается с течением времени** (также известная в определенных контекстах под названием *сглаживающей кривой* (easing curve)). Если вы не указываете функцию расчета времени, то используется функция по умолчанию, а это, в противоположность распространенным ожиданиям, вовсе **не линейная** функция — она показана на рис. 8.2. Обратите внимание на момент, обозначенный точкой на графике: когда **прошла половина времени, отведенного для эффекта, переход уже завершился на 80%!**

Функцию расчета времени по умолчанию также можно явно указать с помощью ключевого слова `ease` — либо в сокращении `animation`/`transition`, либо в свойстве с полным написанием `animation-timing-function`/`transition-timing-function`. Однако так как `ease` — это функция расчета времени по умолчанию, пользы от этого мало. Но существует еще четыре стандартные кривые, с помощью которых вы можете изменить течение анимации; все они показаны на рис. 8.3.

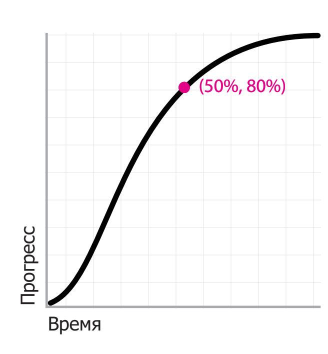


Рис. 8.2. Функция расчета времени по умолчанию (сглаживающая функция) для всех переходов и анимации

Как вы видите, `ease-out` — это противоположность `ease-in`. Это как раз то, что нам требуется для нашего пружинящего эффекта: мы хотим **менять функцию расчета времени на противоположную каждый раз, когда меняется направление движения мячика**. Следовательно, мы можем указать главную функцию расчета времени в свойстве `animation` и переопределять ее в ключевых кадрах. Нам нужно, чтобы основному направлению движения соответствовала функция расчета времени с ускорением (`ease-out`), а обратному — с замедлением (`ease-in`):

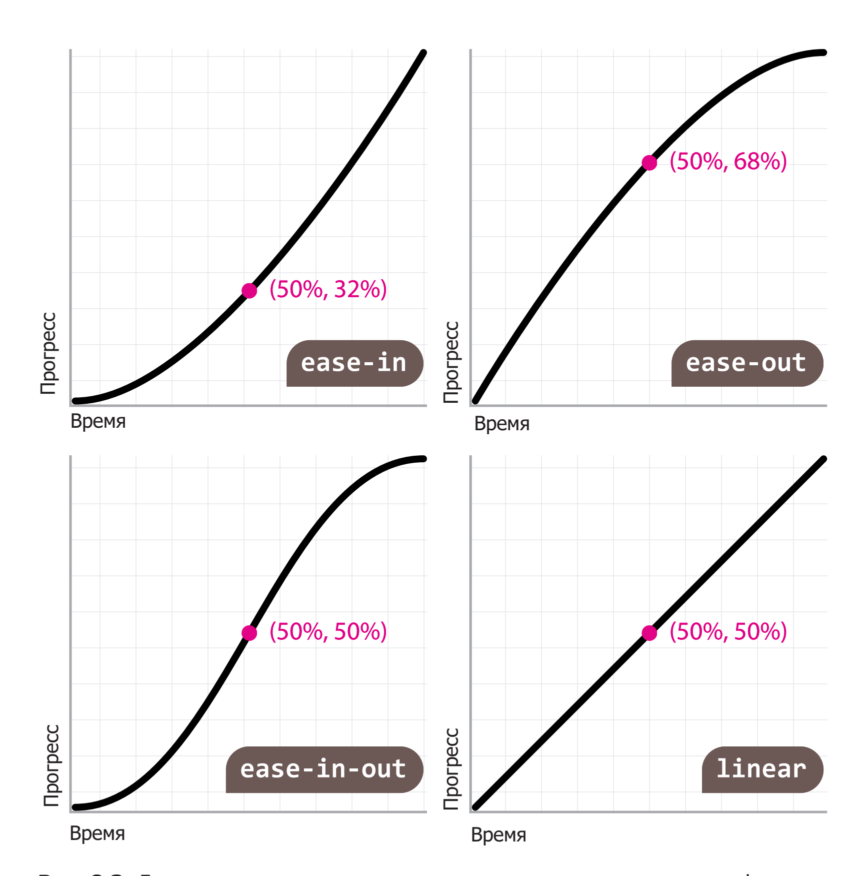


Рис. 8.3. Доступные ключевые слова, соответствующие стандартным функциям расчета времени

```
@keyframes bounce {
    60%, 80%, to {
        transform: translateY(400px);
        animation-timing-function: ease-out;
    }
    70% { transform: translateY(300px); }
    90% { transform: translateY(360px); }
}

.ball {
    /* Остальные стили */
    animation: bounce 3s ease-in;
}
```

Если вы протестируете этот код, то заметите, что даже это простое изменение моментально создает гораздо более реалистичный пружинящий эффект. Однако ограничиваться этими пятью стандартными кривыми чрезвычайно печально. Если бы мы могли использовать произвольные функции расчета

времени, то достигали бы намного более реалистичных результатов. Например, если пружинящая анимация предназначена для иллюстрации падающего объекта, то **более высокое значение ускорения** (такое, какое обеспечивает нам `ease`) создает более реалистичный эффект. Но как нам создать противоположность `ease`, если ключевого слова для этого не предусмотрено?

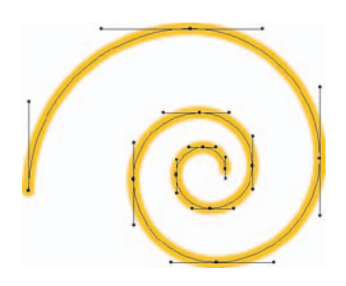


Рис. 8.4. Кубическая кривая Безье для спирали; здесь показаны узловые точки и контрольные точки этой кривой

Все эти пять кривых задаются посредством (кубических) кривых Безье. Кривые Безье — это кривые, с которыми вы работаете в любых приложениях для создания векторной графики (таких, как Adobe Illustrator). Они определяются как наборы сегментов пути с манипулятором на каждом конце, позволяющим управлять их кривизной (эти манипуляторы часто называют *контрольными точками*). Сложные кривые содержат огромное количество подобных сегментов, соединенных своими конечными точками (рис. 8.4). Функции расчета времени CSS — это **кривые Безье с одним только сегментом**, поэтому у них только **две контрольные точки**. В качестве примера на рис. 8.5 показана функция расчета времени по умолчанию (`ease`) и соответствующие контрольные точки.



Рис. 8.5. Функция расчета времени ease со своими узловыми точками и контрольными точками

В дополнение к пяти стандартным кривым, которые мы рассмотрели в предыдущем абзаце, существует также функция `cubic-bezier()`, позволяющая указывать собственные функции расчета времени. Она принимает четыре аргумента, соответствующих координатам двух контрольных точек для необходимой кривой Безье, в формате `cubic-bezier(x1, y1, x2, y2)`, где `(x1, y1)` — это координаты первой контрольной точки, а `(x2, y2)` — координаты второй контрольной точки. Конечные точки сегмента пути фиксированы: в точке `(0, 0)` находится начало перехода (количество прошедшего времени на нуле, прогресс на нуле), а в точке `(1, 1)` — конец перехода (100% времени прошло, 100% прогресса случилось).

Обратите внимание, что наличие одного сегмента с фиксированными конечными точками — не единственное ограничение. Значения координаты `x` обеих контрольных точек ограничены диапазоном [0, 1] (то есть мы не можем вынести манипуляторы за пределы графика по горизонтали). Это ограничение появилось не случайно. Поскольку мы не можем (*пока?*) путешествовать во

времени, невозможно определить переход, начинающийся до того, как он будет запущен, или заканчивающийся после отведенного ему промежутка времени. Но реальное ограничение здесь только одно — это количество узловых точек: возможность определять кривые только с двумя узловыми точками здорово сужает диапазон возможных результатов, но также делает функцию `cubic-bezier()` проще в использовании. Несмотря на эти ограничения, `cubic-bezier()` позволяет создавать весьма широкий диапазон разнообразных функций расчета времени.

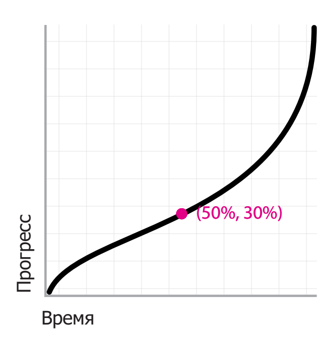


Рис. 8.6. Противоположная функция расчета времени для ease

Отсюда логически вытекает, что мы можем **перевернуть любую функцию расчета времени, поменяв местами горизонтальные координаты с вертикальными в обеих контрольных точках**. Это верно и для ключевых слов; все пять ключевых слов, рассмотренных выше, соответствуют определенным значениям `cubic-bezier()`. Например, `ease` — это эквивалент `cubic-bezier(.25,.1,.25,1)`, поэтому противоположной ей будет `cubic-bezier(.1,.25,1,.25)`. Результат показан на рис. 8.6. Таким образом, теперь в нашей пружинящей анимации мы можем использовать `ease`, и она будет выглядеть еще реалистичнее:

```
@keyframes bounce {
    60%, 80%, to {
        transform: translateY(400px);
        animation-timing-function: ease;
    }
    70% { transform: translateY(300px); }
    90% { transform: translateY(360px); }
}

.ball {
    /* Стилизация */
    animation: bounce 3s cubic-bezier(.1,.25,1,.25);
}
```

Используя графические инструменты, подобные http://cubic-bezier.com (рис. 8.7), мы можем продолжить эксперименты и внести еще больше улучшений в нашу пружинящую анимацию.

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**bounce**

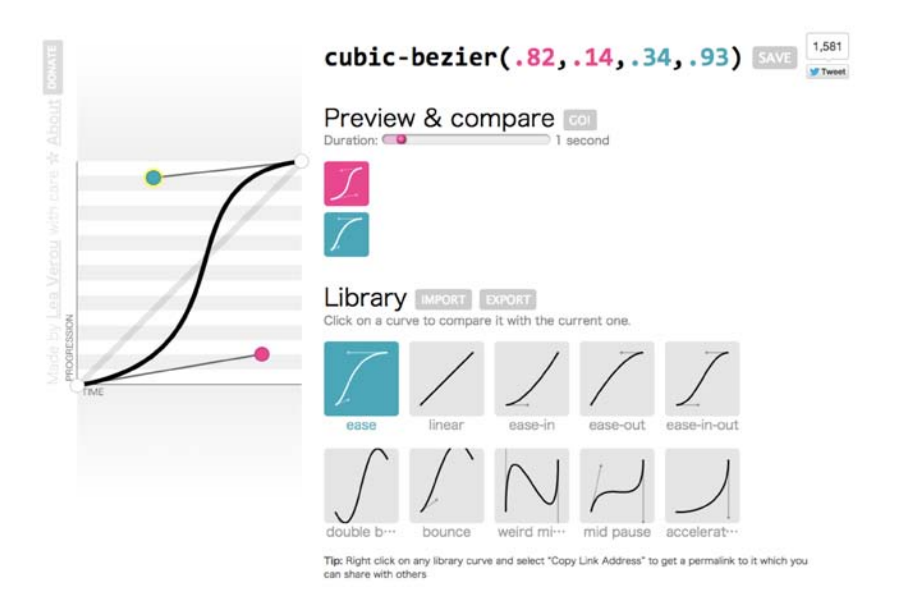


Рис. 8.7. Кубические кривые Безье печально известны сложностью в определении и понимании без соответствующей визуализации, особенно когда они играют роль функций расчета времени для переходов. К счастью, в Сети существуют визуальные инструменты, помогающие разобраться в этом нелегком вопросе, например http://cubic-bezier.com (на рисунке) авторства вашей покорной слуги



В библиотеке анимации animate.css **Дэна Эдена** (Dan Eden, http://daneden.me) используется функция расчета времени `cubic-bezier(.215,.61,.355,1)` и `cubic-bezier(.755,.05,.855,.06)` в качестве противоположной. Противоположная функция характеризуется более крутым графиком, что создает еще более реалистичный эффект.

Эластичные переходы

Предположим, что каждый раз, когда фокус переводится на текстовое поле, мы хотим показывать выноску с дополнительной информацией, например допустимыми значениями для ввода в этом поле. Разметка может выглядеть приблизительно так:

HTML

```
<label>
    Your username: <input id="username" />
    <span class="callout">Only letters, numbers,
    underscores (_) and hyphens (-) allowed!</span>
</label>
```

Рис. 8.8. Как изначально выглядит наш переход

А CSS-код для переключения стиля отображения может выглядеть как в следующем фрагменте (я убрала все относящееся к стилизации и разметке):

```
input:not(:focus) + .callout {
    transform: scale(0);
}

.callout {
    transition: .5s transform;
    transform-origin: 1.4em -.4em;
}
```

СОВЕТ

Если вы использовали для отображения выноски свойство `height`, а не трансформацию, то заметите, что переход от `height: 0` (или любого другого значения) к `height: auto` не работает, так как `auto` — это ключевое слово, и оно не может быть выражено в форме анимируемого значения. В таких случаях следует использовать `max-height` с достаточно большим значением высоты.

В текущем варианте реализации, когда пользователь переводит фокус на наше текстовое поле, запускается переход длительностью 0,5 с, работающий, как показано на рис. 8.8. С ним все прекрасно уже сейчас, но он бы выглядел естественнее и привлекательнее, если бы в конце выноска на мгновение немного раздувалась (то есть увеличивалась до 110% своего размера, а затем снова возвращалась к 100%). Этого можно добиться, преобразовав переход в анимацию и применив трюк, который мы выучили в предыдущем разделе:

```
@keyframes elastic-grow {
    from { transform: scale(0); }
    70% {
        transform: scale(1.1);
        animation-timing-function:
            cubic-bezier(.1,.25,1,.25); /* Обратная к ease */
    }
}

input:not(:focus) + .callout { transform: scale(0); }

input:focus + .callout { animation: elastic-grow .5s; }

.callout { transform-origin: 1.4em -.4em; }
```

Протестировав это решение, мы убедимся, что оно действительно работает. Результат вы можете видеть на рис. 8.9: сравните его с предыдущим вариантом перехода. Но, по сути, мы воспользовались анимацией там, где в действительности нам требовался переход. Анимация — очень мощный инструмент, и в такой ситуации, как у нас, когда мы всего лишь хотим добавить переходу немного эластичности, это все равно что забивать гвозди микроскопом или браться за цепную пилу, чтобы отрезать кусочек хлеба. Можно ли добиться чего-то подобного, используя только переход?

Рис. 8.9. Наш пользовательский интерфейс выглядит реалистичнее и привлекательнее после того, как мы сделали переход немного эластичнее

Решение опять кроется в настраиваемых функциях расчета времени `cubic-bezier()`. Пока мы обсуждали только такие кривые, контрольные точки которых принадлежали диапазону от 0 до 1. Как я уже говорила в предыдущем разделе, нельзя выходить за пределы этого диапазона по горизонтали, хотя в будущем это может измениться, если человечество когда-либо изобретет машину времени. Однако **по вертикали мы можем выходить за рамки диапазона 0–1**, заставляя наш переход демонстрировать **менее 0% прогресса** или **более 100%**. Понимаете, что это означает? Это означает, что при движении от трансформации `scale(0)` к трансформации `scale(1)` мы можем заставить ее перешагнуть финальное значение, достигнув уровня `scale(1.1)` или даже больше, в зависимости от того, насколько крутой мы планируем сделать нашу функцию расчета времени.

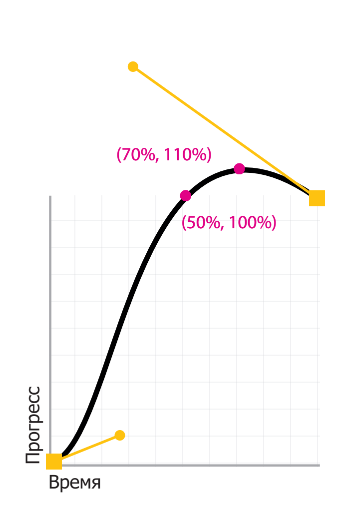


Рис. 8.10. Доработанная функция расчета времени с вертикальными координатами за пределами диапазона 0–1

В нашем случае нужно добавить совсем немного эластичности: мы хотим, чтобы наша функция расчета времени добралась до 110% прогресса (что соответствует `scale(1.1)`), а затем выполнила обратный переход к 100%. Начнем с исходной функции расчета времени `ease` (`cubic-bezier(.25,.1,.25,1)`)

и передвинем вторую контрольную точку наверх, примерно до уровня `bezier(.25,.1,.3,1.5)`. Как видно на рис. 8.10, теперь переход достигает 100% приблизительно через 50% отведенного ему времени. Однако здесь он не останавливается; он продолжает движение вверх, преодолев конечное значение, пока не достигает 110% прогресса приблизительно через 70% времени, а оставшиеся 30% времени занимает возвращение к конечному значению. В результате получается переход, очень схожий с предыдущей анимацией, но для воплощения которого достаточно одной строки кода. Взгляните, как код выглядит теперь:

```
input:not(:focus) + .callout { transform: scale(0); }

.callout {
    transform-origin: 1.4em -.4em;
    transition: .5s cubic-bezier(.25,.1,.3,1.5);
}
```

Однако несмотря на то что, когда мы переводим фокус на текстовое поле, заставляя выноску появляться, наш переход выглядит в точности так, как и ожидалось, в обратной ситуации, когда текстовое поле теряет фокус, а выноска сжимается и исчезает, результат может разочаровывать (рис. 8.11). Что же здесь происходит? Как бы странно этот эффект ни выглядел, ничего неожиданного здесь нет: когда мы переводим фокус с нашего поля ввода на другой элемент интерфейса, запускается переход, начальное значение которого равно `scale(1)`, а конечное — `scale(0)`. Следовательно, поскольку применяется та же самая функция расчета времени, переход все так же достигает 110% прогресса через 350 мс. Только на этот раз 110% прогресса соответствует не `scale(1.1)`, а `scale(-0.1)`!

Рис. 8.11. Что случилось?!

Но не стоит опускать руки, ведь исправить эту проблему можно с помощью всего лишь одной дополнительной строки кода. Предположим, что для эффекта, когда выноска сжимается, нам требуется обычная функция расчета времени `ease`. Для того чтобы добавить ее, нужно всего лишь переопределить текущую функцию расчета времени в правиле CSS, определяющем закрытое состояние:

```
input:not(:focus) + .callout {
    transform: scale(0);
    transition-timing-function: ease;
}

.callout {
    transform-origin: 1.4em -.4em;
    transition: .5s cubic-bezier(.25,.1,.3,1.5);
}
```

Попробуйте выполнить код еще раз, и вы увидите, что выноска закрывается точно так, как это происходило до добавления нашей доработанной функции `cubic-bezier()`, а отображение выноски сопровождается приятным эластичным эффектом.

Самые бдительные читатели наверняка заметили другую проблему: **при закрытии выноски создается впечатление, что это происходит слишком медленно**. Почему так? Давайте поразмышляем. Когда выноска увеличивается, она достигает **100%** конечного размера через **50%** времени (то есть через **250 мс**). Но когда она уменьшается, движение от 0% до 100% занимает **все время**, выделенное для перехода (500 мс), поэтому скорость оказывается **в два раза меньше**.

Чтобы исправить этот недостаток, мы можем переопределить также и длительность перехода, используя `transition-duration` или же сокращение `transition`, которое переопределяет вообще все. Во втором варианте нам не придется явно возвращать функцию расчета времени `ease`, потому что это первоначальное значение:

```
input:not(:focus) + .callout {
    transform: scale(0);
    transition: .25s;
}

.callout {
    transform-origin: 1.4em -.4em;
    transition: .5s cubic-bezier(.25,.1,.3,1.5);
}
```

Хотя эластичность может быть приятным дополнением ко многим типам переходов (некоторые из них я перечислила в разделе «Проблема» этого секрета), **с некоторыми она выглядит просто ужасно**. Типичная ситуация, когда вы **не хотите** использовать эластичные переходы, это работа с **цветами**. Несомненно, эластичные переходы на цветах могут смотреться **довольно забавно** (рис. 8.12), но в пользовательских интерфейсах использовать это чаще всего нежелательно.

Для того чтобы предотвратить ненамеренное применение эластичных переходов к цветам, попробуйте ограничивать переходы определенными свойствами, вместо того чтобы вообще не указывать никакие, как мы делали выше. Если

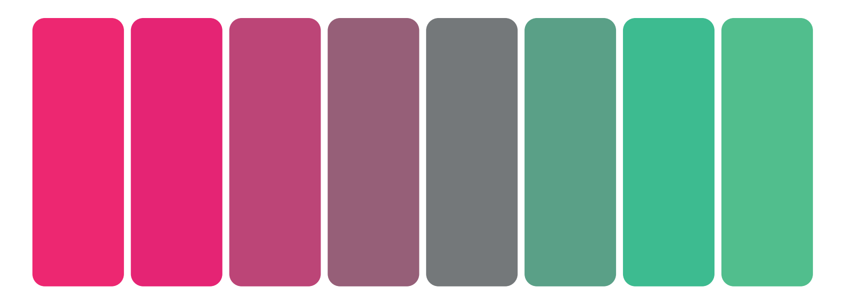


Рис. 8.12. Эластичный цветовой переход от rgb(100%, 0%, 40%) к цвету gray (rgb(50%, 50%, 50%)) с функцией расчета времени cubic-bezier(.25,.1,.2,3). Каждая координата RGB интерполируется по отдельности, поэтому в результате в составе перехода оказываются странные цвета вроде rgb(0%, 100%, 60%). Проверьте на http://play.csssecrets.io/**elastic-color**

в сокращении `transition` мы не указываем никакие свойства, то свойству `transition-property` присваивается значение по умолчанию: `all`. Это означает, что **ко всему, на что может распространяться переход, этот переход будет применен**. Следовательно, если позднее к правилу, в котором описываются открывающиеся выноски, мы добавим изменение фона в свойстве `background`, то эластичный переход будет применен также и к установке фона. Финальная версия кода выглядит так:

```
input:not(:focus) + .callout {
    transform: scale(0);
    transition: .25s transform;
}

.callout {
    transform-origin: 1.4em -.4em;
    transition: .5s cubic-bezier(.25,.1,.3,1.5)
transform;
}
```

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**elastic**

СОВЕТ

Продолжая тему ограничения переходов конкретными свойствами, вы можете даже ставить в очередь переходы, определенные для разных свойств, используя свойство `transition-delay` — второе временнóе значение в сокращении `transition`. Например, если переход охватывает оба атрибута, `width` и `height`, но вы хотите, чтобы сначала изменилась высота и только после этого ширина (эффект, ставший популярным благодаря множеству сценариев для реализации световых коробов), то можете использовать подобное правило: `transition: .5s height, .8s .5s width;` (то есть задержка перехода для атрибута `width` равна продолжительности перехода для атрибута `height`).

СВЯЗАННЫЕ СПЕЦИФИКАЦИИ

CSS Transitions: http://w3.org/TR/css-transitions

CSS Animations: http://w3.org/TR/css-animations

43 Покадровая анимация

ПРЕДВАРИТЕЛЬНЫЕ ТРЕБОВАНИЯ

Базовые знания анимации CSS, секрет «Эластичные переходы»

Проблема

Довольно часто возникает необходимость в анимации, которую трудно или даже вовсе невозможно создать с помощью переходов, определенных на CSS-свойствах элементов. Например, это может быть мультфильм или сложный индикатор прогресса. Покадровая анимация, состоящая из отдельных изображений, — идеальное решение для подобных случаев, но удивительно, насколько сложно реализовать ее гибким и удобным способом.

Возможно, вы сейчас задаетесь вопросом, почему бы просто не прибегнуть к помощи анимированных изображений в формате GIF. Разумеется, анимированные GIF-изображения прекрасно подходят во многих ситуациях, но у них есть несколько недостатков, которые в определенных сценариях делают их нежелательным решением:

- они ограничены **палитрой из 256 цветов, одной и той же для всех кадров**;
- они не поддерживают **прозрачность альфа-канала**, что может стать большой проблемой, если мы не знаем, что будет находиться под нашим анимированным GIF-изображением. Например, так часто бывает, когда изображение представляет собой индикатор прогресса (рис. 8.13);
- нет никакой возможности управлять из CSS-кода определенными аспектами, такими как продолжительность, количество повторений, приостановка анимации и т. д. Генерируя изображение в формате GIF, вы пакуете в один

файл все данные, и изменить их можно, лишь отредактировав изображение и создав новый файл. Это прекрасно для **обеспечения переносимости, но не когда вы хотите поэкспериментировать**.

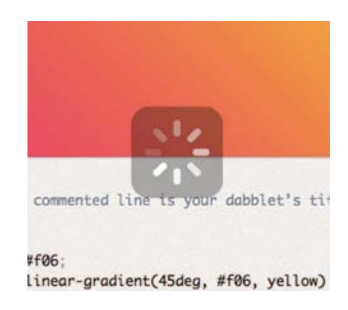


Рис. 8.13. Полупрозрачный индикатор прогресса (на веб-сайте http://www.dabblet.com); такого результата невозможно добиться с помощью анимированных изображений в формате GIF

Давным-давно, в 2004 году, разработчики браузера Mozilla предприняли попытку справиться с первыми двумя проблемами, разрешив **покадровую анимацию в файлах формата PNG**, аналогично тому, как мы можем использовать и статичные, и анимированные GIF-файлы. Формат носил название *APNG* и предусматривал обратную совместимость с утилитами просмотра изображений, не поддерживающими PNG с анимацией: первый кадр кодировался точно так же, как в традиционных PNG-файлах, поэтому старые утилиты могли как минимум отобразить один этот кадр. Несмотря на многообещающее начало, формат APNG так и не завоевал популярность, и по сей день поддержка этого формата браузерами и графическими редакторами крайне ограниченна.

Разработчики даже используют JavaScript для реализации гибкой покадровой анимации в браузере, анимируя с помощью сценариев свойство `background-position` спрайта. Более того, в Сети можно найти небольшие библиотеки, предназначенные для помощи в этом деле! Но существует ли простой и понятный способ создания подобной анимации средствами приятного и читабельного CSS-кода?

Получить более подробную информацию о формате APNG вы можете в статье https://ru.wikipedia.org/wiki/APNG.

Решение

Предположим, что у нас есть спрайт в формате PNG, содержащий все кадры нашей анимации, как показано на рис. 8.14.



Рис. 8.14. Восемь кадров нашего индикатора прогресса (размер спрайта — 800×100)

Также у нас есть элемент, который будет содержать этот индикатор (не забудьте для обеспечения доступности добавить описательный текст!), и мы определили для него размеры, совпадающие с размерами одного кадра:

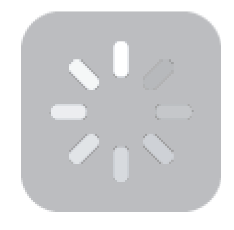


Рис. 8.15. Первый кадр нашего индикатора загрузки отображается, но пока анимация отсутствует

HTML

```
<div class="loader">Loading…</div>

.loader {
    width: 100px; height: 100px;

    background: url(img/loader.png) 0 0;

    /* Скрыть текст */
    text-indent: 200%;
    white-space: nowrap;
    overflow: hidden;
}
```

Пока результат выглядит как на рис. 8.15: первый кадр отображается, но никакой анимации не видно. Однако если мы воспроизведем код с разными значениями `background-position`, то заметим, что `-100px 0` дает нам второй кадр, `-200px 0` — третий кадр и т. д. Первой мыслью в связи с этим может быть такой вариант анимации:

```
@keyframes loader {
    to { background-position: -800px 0; }
}

.loader {
    width: 100px; height: 100px;
    background: url(img/loader.png) 0 0;
    animation: loader 1s infinite linear;

    /* Скрыть текст */
    text-indent: 200%;
    white-space: nowrap;
    overflow: hidden;
}
```

Однако, как вы можете видеть на следующих снимках экрана (сделанных с интервалом 167 мс), это решение не работает (рис. 8.16).

Вам может казаться, что мы зашли в тупик, но в действительности мы очень близки к решению. Секрет здесь заключается в использовании функции расчета времени `steps()` вместо функции, основанной на кривых Безье.

«Какой-какой функции расчета времени?!» — спросите вы. Как мы узнали в предыдущем разделе, все функции расчета времени на основе кривых Безье интерполируют содержимое ключевых кадров, для того чтобы обеспечить плавный переход одного изображения в другое. Это великолепно; чаще всего плавное перетекание и есть та причина, по которой мы прибегаем к переходам и анимации CSS. Однако в нашей ситуации **плавность разрушает нашу анимацию спрайта**.

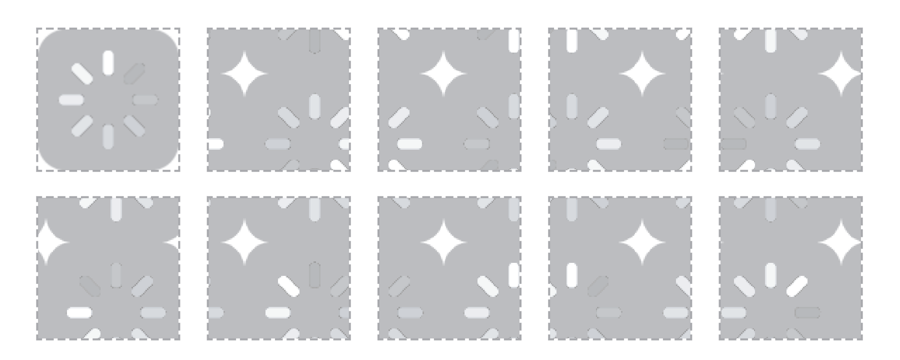


Рис. 8.16. Наша первая попытка реализовать покадровую анимацию провалилась, так как в действительности нам не требуются плавные переходы между ключевыми кадрами

В отличие от функций расчета времени Безье, функция `steps()` делит всю анимацию на кадры по количеству указанных вами шагов и резко перескакивает с одного на другой безо всякой интерполяции. Обычно такая резкая смена картинки нежелательна, поэтому о `steps()` вспоминают крайне редко. В мире функций расчета времени CSS функции на основе кривых Безье — это популярные ребята, которых приглашают на все вечеринки, а `steps()`, к сожалению, — гадкий утенок, с которым никто не хочет даже пообедать. Но в нашей задаче требуется именно такое поведение. Как только мы переформулируем определение анимации показанным далее способом, индикатор загрузки сразу же начнет работать так, как и планировалось с самого начала:

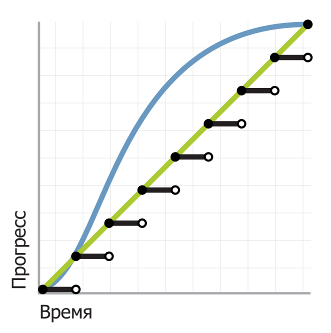


Рис. 8.17. Сравнение функций расчета времени steps(8), linear и функции по умолчанию ease

```
animation: loader 1s infinite steps(8);
```

Помните, что `steps()` также принимает необязательный второй параметр, `start` или `end` (значение по умолчанию), определяющий, когда в каждом интервале

происходит переключение (поведение по умолчанию для варианта `end` показано на рис. 8.17), но необходимость в нем возникает крайне редко. Если вам нужен только один шаг, то можно воспользоваться одним из сокращений — `step-start` или `step-end`, которые эквивалентны `steps(1, start)` и `steps(1, end)` соответственно.

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**frame-by-frame**


Благодарности

Благодарю **Simurai** (http://simurai.com) за публикацию описания этой полезной техники в статье **Sprite sheet animation with steps()** (http://simurai.com/blog/2012/12/03/step-animation).

СВЯЗАННЫЕ СПЕЦИФИКАЦИИ

CSS Animations: http://w3.org/TR/css-animations

44 Мерцание

ПРЕДВАРИТЕЛЬНЫЕ ТРЕБОВАНИЯ

Базовые знания об анимации CSS, секрет «Покадровая анимация»

Проблема

Помните старый тег `<blink>`? Разумеется, помните. Он превратился в культурный символ нашей индустрии, напоминающий нам о неловком, непритязательном начале нашей дисциплины, а также в семейную шутку корифеев веб-дизайна. Он презираем всюду и всеми за нарушение правила о разделении структуры и стиля, но в первую очередь за чрезмерное использование в конце девяностых, из-за чего работа в Сети в то время была сущим кошмаром. Даже его создатель Лу Монтулли признался: «[Я считаю] мерцающий тег худшим, что я когда-либо сделал для Интернета».

Однако теперь, когда раны, нанесенные тегом `<blink>`, затянулись, мы иногда ловим себя на мысли о добавлении в дизайн мерцающей анимации. Поначалу это кажется чем-то странным, словно мы обнаруживаем в себе жуткие наклонности, о которых до этого не подозревали. Но кризис самопознания проходит, когда мы осознаем, что в некоторых редких сценариях мерцание может **улучшить впечатление пользователя от использования вашего дизайна, а не ухудшить его**.

Распространенный прием в дизайне пользовательских интерфейсов — добавление мерцания (но не более трех вспышек!) для указания, что определенное изменение было применено к интерфейсу, или же для подсветки цели текущей ссылки (элемента, чей идентификатор совпадает с указанным в адресе после решетки `#`). При таком ограниченном использовании мерцание может быть очень эффективным инструментом привлечения внимания пользователя к определенной области экрана, а благодаря малому количеству итераций мы избегаем побочных эффектов, которые порождал тег `<blink>`. Еще один трюк, позволяющий использовать мерцание во благо (для управления вниманием пользователя) и при этом не наносить ущерб (не отвлекать, не раздражать и не

вызывать припадков), заключается в том, чтобы «сглаживать» вспышки (то есть вместо простого переключения между состояниями «включено» и «выключено» мы добавляем плавный переход от одного к другому).

Но как же реализовать все это? Единственная подходящая замена тегу `<blink>` в CSS, свойство `text-decoration: blink`, слишком ограниченно и не позволяет воплощать все задумки, но даже если бы оно обладало достаточной мощью, все равно поддержка браузерами у него находится на очень низком уровне. Так можем ли мы использовать CSS или наша единственная надежда — JS?

Решение

В действительности реализовать подобное мерцание с помощью анимации CSS можно несколькими способами: анимировать весь элемент (свойство `opacity`), анимировать цвет текста (свойство `color`), анимировать его рамку (свойство `border-color`) и т. д. Далее в этом разделе мы будем предполагать, что в нашем решении должен мерцать только текст, так как это самый распространенный сценарий использования. Для других составляющих элемента решение будет аналогичным.

Добиться плавного мерцания довольно просто. Наша первая попытка могла бы выглядеть так:

```
@keyframes blink-smooth { to { color: transparent } }

.highlight { animation: 1s blink-smooth 3; }
```

Это почти сработало. Наш текст плавно теряет цвет, уходя в полную прозрачность, но затем **внезапно снова полностью восстанавливает исходный цвет**. Иллюстрация изменения цвета текста с течением времени наглядно показывает, почему так происходит (рис. 8.18):

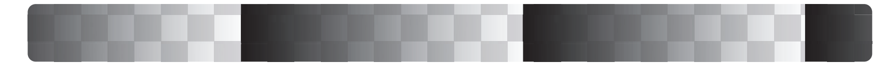


Рис. 8.18. Изменение цвета текста на протяжении 3 с (три итерации)

В действительности иногда именно такой эффект нам и требуется. Если в вашем случае это так, то задача решена! Однако если мы хотим, чтобы мерцание было плавным и при переходе к прозрачности, и при восстановлении цвета, нам нужно еще немного потрудиться. Один из способов добиться этого — поменять ключевые кадры, для того чтобы переключение происходило в середине каждой итерации:

```
@keyframes blink-smooth { 50% { color: transparent } }

.highlight {
    animation: 1s blink-smooth 3;
}
```

Кажется, мы достигли желаемого результата. Однако даже если это не заметно на данной конкретной анимации (потому что на переходах цвета/прозрачности различия между функциями расчета времени не так видны), важно помнить, что анимация ускоряется как при пропадании, так и при наборе цвета, что в определенных сценариях (например, когда мы создаем пульсирующую анимацию) выглядит неестественно. Для того случая у нас в рукаве припасен еще один козырь: `animation-direction`.

Единственное предназначение `animation-direction` — менять направление анимации либо во всех итерациях (`reverse`), либо в каждой четной итерации (`alternate`), либо в каждой нечетной итерации (`alternate-reverse`). Лучше всего здесь то, что при этом **функция расчета времени также меняется на обратную**, создавая гораздо более реалистичную анимацию. Попробуем применить это к нашему мерцающему элементу:

```
@keyframes blink-smooth { to { color: transparent } }

.highlight {
    animation: .5s blink-smooth 6 alternate;
}
```

Обратите внимание, что нам пришлось удвоить количество итераций (а не их продолжительность, как в предыдущем способе), поскольку теперь каждая пара из проявления и исчезновения цвета состоит из двух итераций. По той же причине мы вполовину уменьшили `animation-duration`.

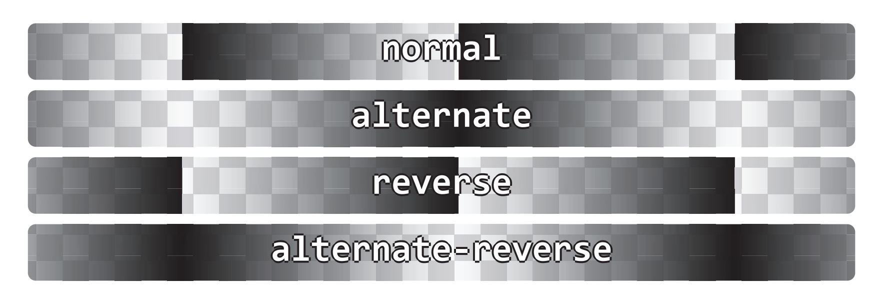


Рис. 8.19. Все четыре значения animation-direction и как под их влиянием на протяжении трех итераций происходит процесс перехода цвета от black до transparent

Если мы добивались плавной мерцающей анимации, то дело сделано. Но что, если нам требуется классическое решение? Как быть? Первая попытка может выглядеть примерно так:

```
@keyframes blink { to { color: transparent } }

.highlight {
    animation: 1s blink 3 steps(1);
}
```

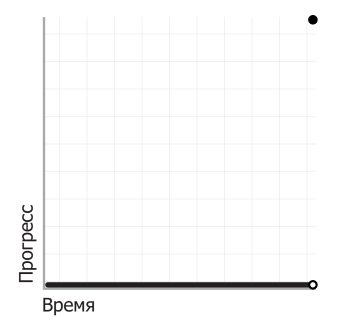


Рис. 8.20. Что в действительности делает функция расчета времени steps(1) с нашей анимацией

Однако попытка выполнить этот код ведет к полнейшему провалу: абсолютно ничего не происходит. Причина в том, что `steps(1)` по сути — это эквивалент `steps(1, end)`, что означает, что переход между текущим цветом и цветом `transparent` происходит за один шаг, а **значение переключается в конце** (рис. 8.20). Следовательно, **мы видим начальное значение на протяжении всей анимации, за исключением бесконечно короткого промежутка в самом конце**. Если мы поменяем функцию расчета времени на `steps(1, start)`, то произойдет ровно противоположное: переключаться цвет будет в начале, поэтому мы будем видеть только прозрачный текст безо всякой анимации или мерцания.

На следующем шаге логично испробовать `steps(2)` в обоих вариантах (с ключевыми словами `start` и `end`). Теперь мы видим какое-то мерцание, но оно происходит между полупрозрачным и прозрачным текстом или полупрозрачным текстом и текстом обычного цвета — соответственно по той же причине. К сожалению, поскольку невозможно настроить `steps()` так, чтобы переключение происходило в середине — только в начале или в конце, — единственным решением здесь будет скорректировать ключевые кадры анимации, переместив точку переключения на 50%, как мы уже делали раньше:

```
@keyframes blink { 50% { color: transparent } }

.highlight {
    animation: 1s blink 3 steps(1); /* или step-end */
}
```

Наконец-то все заработало! Кто бы мог подумать, что классическое прерывистое мерцание будет реализовать сложнее, чем современное плавное? CSS никогда не перестает удивлять...

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**blink**

СВЯЗАННЫЕ СПЕЦИФИКАЦИИ

CSS Animations: http://w3.org/TR/css-animations

45 Имитация ввода текста

ПРЕДВАРИТЕЛЬНЫЕ ТРЕБОВАНИЯ

Базовые знания анимации CSS, секрет «Покадровая анимация», секрет «Мерцание»

Проблема

Иногда перед нами встает задача создать эффект текста, появляющегося на экране символ за символом, словно кто-то его вводит. Этот прием в сочетании с моноширинными шрифтами особенно часто используется на технических веб-сайтах для имитации командной строки терминала. При правильном использовании он способен прекрасно вписаться и значительно улучшить общий дизайн страницы.

Обычно для реализации такого эффекта разработчикам приходится прибегать к помощи длинного, грязного и сложного JS-кода. И несмотря на то что это всего лишь представление, использование CSS для подобного эффекта кажется недостижимой мечтой. Или все же это возможно?

Решение

Основная идея заключается в том, чтобы **анимировать ширину** элемента, содержащего наш текст, с нуля и до полного значения, увеличивая размер видимой области на один символ за раз. Вероятно, вы уже догадались, какое ограничение накладывает данный подход: **для многострочного текста он не работает**. К счастью, чаще всего подобная стилизация требуется для однострочного текста, такого как заголовки.



Рис. 8.21. Мы использовали вариацию такого анимационного эффекта в CERN для создания веб-страницы, имитирующей первый браузер, работающий в строковом режиме (http://line-mode.cern.ch)

Теоретически мы могли бы использовать это решение и с многострочным текстом, но для этого потребовалось бы обернуть каждую строку в собственный элемент и обеспечить необходимую задержку старта анимации (то есть решение в итоге получится хуже изначальной проблемы).

Еще одна вещь, о которой не следует забывать, — **каждая анимация по мере увеличения длительности порождает все больший снижающий эффект**: короткая анимация делает интерфейс более стильным и в некоторых случаях даже способна повысить удобство использования. Но чем длиннее анимация, тем скорее она начинает раздражать пользователя. Следовательно, **даже если эту технику можно было бы применить к длинному многострочному тексту, в большинстве случаев это было бы очень плохой идеей**.

CSS is awesome!

Рис. 8.22. Наша отправная точка

Приступим к кодированию! Предположим, что мы хотим применить этот эффект к заголовку верхнего уровня (`<h1>`), для которого мы уже определили стиль, включающий моноширинный шрифт, и который выглядит так:

HTML

```
<h1>CSS is awesome!</h1>
```

Мы можем с легкостью добавить анимацию, меняющую ширину нашего заголовка от 0 до конечного значения, как показано далее:

```
@keyframes typing {
    from { width: 0 }
}

h1 {
    width: 7.7em; /* Ширина текста */
    animation: typing 8s;
}
```

CSS
is
awesome!

CSS is
awesome!

CSS is awesome!

Рис. 8.23. Наша первая попытка создать анимацию, имитирующую ввод текста, на ввод текста совершенно не похожа

Все правильно, ведь так? Однако, как вы видите на рис. 8.23, у этого безобразия нет ничего общего с эффектом, которого мы хотели добиться.

Наверное, вы уже догадались, в чем проблема. Во-первых, мы забыли применить `white-space: nowrap;` для предотвращения переноса текста на новую строку, поэтому по мере увеличения ширины элемента количество строк меняется. Во-вторых, мы забыли применить `overflow: hidden;`, поэтому обрезания не происходит. Исправив эти недочеты, мы начинаем видеть реальные проблемы нашей анимации (рис. 8.24), а именно:

CSS

CSS is aw

CSS is aweson

Рис. 8.24. Наша вторая попытка ближе к желаемой цели, но это все еще не окончательное решение

- очевидно, что анимация получается плавной, вместо того чтобы отображать текст символ за символом;
- менее очевидная проблема заключается в том, что до сих пор мы указывали ширину в единицах измерения `em`, что, конечно, лучше, чем делать это в пикселах, но тоже не идеально. Откуда взялось значение 7,7? Как его вычислить?

Первую проблему можно решить с помощью функции расчета времени `steps()`, как мы делали в секретах «Покадровая анимация» и «Мерцание». К сожалению, требуемое количество шагов равно количеству символов в нашей строке, а значит, такой код будет сложно поддерживать, а в случае динамического текста — попросту невозможно создать. Однако, как мы увидим чуть далее, вычисление этого значения можно автоматизировать, используя крошечный фрагмент кода JavaScript.

Обойти вторую проблему нам помогут единицы измерения `ch`. `ch` — это одна из новых единиц измерения, которая была добавлена в спецификации **CSS**

CS

CSS is a

CSS is aweso

Рис. 8.25. Теперь текст отображается символ за символом, но чего-то еще не хватает

Values and Units Level 3 (http://w3.org/TR/css3-values) и представляет ширину глифа 0. Это одна из наименее известных новых единиц измерения, так как в большинстве случаев нас мало волнует возможность определять размеры элементов относительно ширины глифа 0. Однако моноширинные шрифты — особый случай. **В моноширинных шрифтах ширина глифа 0 совпадает с шириной любого другого глифа.** Следовательно, для того чтобы задать ширину в единицах измерения `ch`, нужно указать количество символов. В нашем примере их `15`. Соберем все вместе:

```
@keyframes typing {
    from { width: 0; }
}

h1 {
    width: 15ch; /* Ширина текста */
    overflow: hidden;
    white-space: nowrap;
    animation: typing 6s steps(15);
}
```

Как подтверждают кадры, показанные на рис. 8.25, теперь наконец-то мы получили желаемый результат: наш текст отображается символ за символом. Однако он все еще не выглядит достаточно реалистичным. Догадаетесь, чего не хватает?

Последняя деталь, которая сделала бы наше решение намного реалистичнее, — это **мигающий курсор**. Мы уже научились создавать мерцающую анимацию в секрете «Мерцание». В данном случае курсор можно добавить посредством псевдоэлемента, используя свойство `opacity` для реализации мерцания. С другой стороны, мы могли бы сэкономить псевдоэлемент, которых у нас и так не очень много, на случай, если он понадобится для другого эффекта, и сделать мигающий курсор из правой рамки:

```
@keyframes typing {
    from { width: 0 }
}

@keyframes caret {
    50% { border-color: transparent; }
}

h1 {
    width: 15ch; /* Ширина текста */
```

```
    overflow: hidden;
    white-space: nowrap;
    border-right: .05em solid;
    animation: typing 6s steps(15),
               caret 1s steps(1) infinite;
}
```

Обратите внимание, что, в отличие от анимационного эффекта для отображения текста, курсор должен мерцать бесконечно (даже после того, как мы показали весь текст), отсюда и ключевое слово `infinite`. Помимо этого, мы не указываем цвет рамки, так как он должен определяться автоматически и совпадать с цветом текста. Несколько кадров из результирующей анимации показаны на рис. 8.26.

CS

CSS is a

CSS is aweso

Рис. 8.26. Теперь нашу анимацию дополняет реалистичный мигающий курсор

Наша анимация работает просто великолепно, но проблему сопровождения этого кода мы пока не решили: для каждого заголовка необходимо определять собственный стиль, зависящий от количества символов, и обновлять значения каждый раз, когда мы меняем что-то в содержимом. Это как раз та задача, с которой превосходно способен справиться JS:

JS

```
$$('h1').forEach(function(h1) {
    var len = h1.textContent.length, s = h1.style;

    s.width = len + 'ch';
    s.animationTimingFunction = "steps("+len+"),steps(1)";
});
```

Эти несколько строк кода JS позволяют нам успешно догнать и схватить обоих зайцев: наша анимация не только реалистично выглядит, но и проста в сопровождении!

Все это прекрасно, но что произойдет, если окажется, что браузер не поддерживает анимацию CSS? По сути, такой браузер пропустит весь код, как-либо связанный с анимацией, и прочитает только следующее:

```
h1 {
    width: 15ch; /* Ширина текста */
    overflow: hidden;
    white-space: nowrap;
    border-right: .05em solid;
}
```

В зависимости от того, поддерживает ли такой браузер единицы измерения `ch`, пользователь увидит один из запасных вариантов, показанных на рис. 8.27. Для того чтобы избежать второго случая, можно добавить резервное решение с единицами измерения `em`. Если же вы не хотите видеть немерцающий курсор в резервном решении, то измените анимацию курсора, добавив в ключевые кадры рамку, для того чтобы в случае отсутствия анимации отображалась только невидимая прозрачная рамка, вот так:

CSS is awesome!

CSS is awesome!

Рис. 8.27. Возможные варианты отображения в браузерах, не поддерживающих анимацию CSS (*наверху*: с поддержкой единиц измерения ch; *внизу*: без поддержки единиц измерения ch)

```
@keyframes caret {
    50% { border-color: currentColor; }
}

h1 {
    /* ... */
    border-right: .05em solid transparent;
    animation: typing 6s steps(15),
               caret 1s steps(1) infinite;
}
```

Это достаточно хорошее резервное решение: в старых браузерах анимации нет, но ничего не ломается и текст остается доступным, в то же время не теряя в стилизации.

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**typing**

СВЯЗАННЫЕ СПЕЦИФИКАЦИИ

CSS Animations: http://w3.org/TR/css-animations

CSS Values & Units: http://w3.org/TR/css-values

46 Плавная анимация состояния

ПРЕДВАРИТЕЛЬНЫЕ ТРЕБОВАНИЯ

Базовые знания анимации CSS, свойство `animation-direction` (упоминалось в секрете «Мерцание»)

Проблема

Анимация не всегда запускается сразу же по завершении загрузки страницы. Гораздо чаще мы хотим использовать **анимацию в ответ на действия пользователя**, такие как наведение мыши на элемент или удержание кнопки мыши, когда ее указатель находится на элементе (`:active`). В подобной ситуации у нас не всегда есть контроль над фактическим числом итераций, так как действия пользователя могут приводить к тому, что анимация прервется до того, как будут воспроизведены все запланированные итерации. Например, пользователь может запустить симпатичную анимацию `:hover` и убрать указатель мыши с элемента до того, как анимация завершится. Что, по вашему мнению, должно происходить далее в этом случае?

Если вы ответили что-то вроде «анимация должна застыть в текущем состоянии» или «она должна плавно вернуться в изначальное состояние», то вас ждет неприятный сюрприз. По умолчанию анимация **прервется и резко перескочит в изначальное состояние**. Иногда, если речь идет о малозаметной анимации, такое поведение может быть допустимым. Но в большинстве случаев это значительно ухудшает впечатление пользователей от работы с интерфейсом. Можно ли это как-нибудь изменить?

Это еще одна причина, для того чтобы по возможности пользоваться переходами. Вместо того чтобы резко перепрыгивать к изначальному состоянию, переходы воспроизводятся в обратном направлении, обеспечивая плавное возвращение в исходное состояние.





Рис. 8.28. Я поставила себе целью найти решение этой проблемы, когда работала над простым одностраничным веб-сайтом в подарок на день рождения моему другу Джулиану (Julian, http://juliancheal.co.uk). Обратите внимание на круглое изображение справа на снимке экрана. В действительности я использовала файл в альбомной ориентации. Окружность отрезает правую часть изображения, но когда пользователь наводит на него указатель мыши, изображение начинает медленно прокручиваться влево, и пользователь видит скрытую часть. По умолчанию, если пользователь убирает указатель мыши с изображения, оно резко возвращается в исходную позицию, из-за чего создается впечатление, что с дизайном что-то не так. Поскольку это крошечный веб-сайт, а изображение — его центральный элемент, я решила, что не могу оставить все как есть

Решение

Предположим, что у нас есть очень длинная фотография в альбомной ориентации, такая, как показана на рис. 8.29, но размер доступного пространства, на котором мы можем ее отображать, ограничивается 150 × 150 пикселов. Один



Рис. 8.29. Файл naxos-greece.jpg, который мы будем использовать в примерах в этом секрете, целиком (фотография сделана Крисом Хатчисоном)

из способов решить проблему отображения — добавить анимацию: показывать по умолчанию левый край и прокручивать изображение, открывая оставшуюся часть, когда пользователь взаимодействует с ним (например, наводит на него указатель мыши). Мы будем использовать для изображения один элемент, а анимировать будем позицию его фона:

```
.panoramic {
    width: 150px; height: 150px;
    background: url("img/naxos-greece.jpg");
    background-size: auto 100%;
}
```

Сейчас результат выглядит, как показано на рис. 8.29, и никакой анимации или интерактивности здесь нет. Но если мы поэкспериментируем, то заметим, что при изменении значения `background-position` вручную с исходных `0 0` до `100% 0` будет происходить прокрутка всего изображения. Мы только что нашли наши ключевые кадры!

```
@keyframes panoramic {
    to { background-position: 100% 0; }
}

.panoramic {
    width: 150px; height: 150px;
    background: url("img/naxos-greece.jpg");
    background-size: auto 100%;
    animation: panoramic 10s linear infinite alternate;
}
```

Это решение прекрасно работает. Чем-то напоминает панорамный вид, как будто вы находитесь прямо там и смотрите направо и налево. Однако анимация



Рис. 8.30. Наше изображение обрезано

запускается при загрузке страницы, что может слишком сильно **отвлекать** внимание пользователя в контексте, например, веб-сайта с советами путешественнику, когда он пытается сфокусироваться и прочитать описание Наксоса, вместо того чтобы разглядывать прелестную панорамную фотографию. Гораздо лучше было бы, если бы **анимация стартовала, только когда пользователь наводит на изображение указатель мыши**. Нашей первой мыслью было бы такое изменение кода:

```
.panoramic {
    width: 150px; height: 150px;
    background: url("img/naxos-greece.jpg");
    background-size: auto 100%;
}

.panoramic:hover, .panoramic:focus {
    animation: panoramic 10s linear infinite alternate;
}
```

Это действительно работает, как и ожидается, когда мы наводим указатель мыши на изображение: оно начинает медленно прокручиваться, и мы получаем возможность увидеть его правую часть. Однако когда мы убираем с него указатель мыши, оно резко возвращается к изначальному состоянию, когда на экране виден только его левый край (рис. 8.31). Мы натолкнулись на проблему, которой и посвящен этот секрет!



Рис. 8.31. При наведении указателя мыши фотография начинает плавно двигаться, но когда мы убираем указатель, это приводит к резкой смене картинки, создавая впечатление, что элемент сломан

Для того чтобы исправить это, необходимо посмотреть на желаемый результат под другим углом. Мы не должны ставить себе целью запускать анимацию по событию `:hover`, ведь при этом предыдущее положение не запоминается. Нам

нужно **приостанавливать анимацию, когда событие `:hover` не происходит**. К счастью, существует свойство, назначение которого как раз и состоит в приостановке существующей анимации: `animation-play-state`!

Следовательно, мы применим нашу исходную анимацию к `.panoramic`, но с самого начала поставим ее на паузу до тех пор, пока не сработает событие `:hover`. Поскольку теперь суть решения не в том, чтобы запускать и отменять анимацию, а всего лишь в том, чтобы приостанавливать и продолжать существующую анимацию, резкой перемотки назад не происходит. Финальная версия кода представлена далее, а результат вы можете видеть на рис. 8.32:

Рис. 8.32. Теперь, когда мы убираем указатель мыши с картинки, анимация просто приостанавливается — больше никаких резких прыжков

```
@keyframes panoramic {
    to { background-position: 100% 0; }
}

.panoramic {
    width: 150px; height: 150px;
    background: url("img/naxos-greece.jpg");
    background-size: auto 100%;
    animation: panoramic 10s linear infinite alternate;
    animation-play-state: paused;
}

.panoramic:hover, .panoramic:focus {
    animation-play-state: running;
}
```

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**state-animations**

СВЯЗАННЫЕ СПЕЦИФИКАЦИИ

CSS Animations: http://w3.org/TR/css-animations

47 Анимация вдоль окружности

ПРЕДВАРИТЕЛЬНЫЕ ТРЕБОВАНИЯ

Анимация CSS, трансформации CSS, секрет «Параллелограммы», секрет «Изображения в форме ромба», секрет «Мерцание»

Проблема

Несколько лет назад, когда анимация CSS была нам еще в новинку, Крис Койер (http://css-tricks.com) спросил меня, могу ли я придумать способ анимировать с помощью CSS движение элемента по кругу. В то время это было всего лишь занимательным упражнением на знание CSS, но позднее мне довелось натолкнуться на множество реальных сценариев использования. Например, в Google+ вы видите такую анимацию, когда в круг, в котором уже есть более 11 членов, добавляется новый пользователь: существующие аватары раздвигаются, освобождая место на окружности для нового изображения.

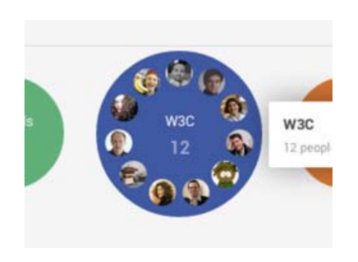


Рис. 8.33. В Google+ анимация вдоль окружности используется для указания, что в «круг» был добавлен новый пользователь

Другой, очень забавный пример можно найти на популярном российском веб-сайте habrahabr.ru (рис. 8.34). В соответствии с лучшими практиками оформления страниц для ошибки 404 эта страница содержит навигационное меню, позволяющее перейти к некоторым основным разделам веб-сайта.

Каждый элемент меню представлен в виде планеты, вращающейся по окружности, а текст наверху гласит: «Слетайте на другие наши планеты». Разумеется, в данном случае логично перемещать планеты по окружности и не вращать их дополнительно вокруг своей оси, иначе текст станет невозможно прочитать.



Рис. 8.34. Страница ошибки 404 популярного российского веб-сайта habrahabr.ru

Это лишь пара из множества подобных примеров. Но как реализовать такой эффект с помощью анимации CSS?

Мы будем работать над очень простым примером аватара, движущегося по окружности, — что-то вроде упрощенной версии упомянутого выше эффекта из Google+. Разметка выглядит так:

Если вы сомневаетесь в своем умении создавать круглые *фигуры* с помощью CSS, то обратитесь к секрету «Гибкие эллипсы».

HTML

```
<div class="path">
    <img src="lea.jpg" class="avatar" />
</div>
```

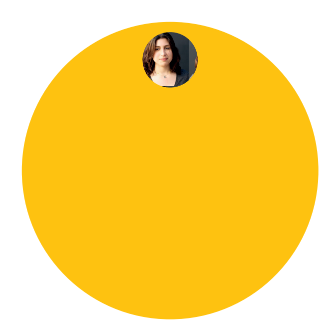


Рис. 8.35. Наша отправная точка после определения нескольких простейших стилей — теперь мы можем полностью отдаться анимации CSS!

Прежде чем задумываться об анимации, необходимо применить несколько базовых стилей (определить размеры, фоны, поля и т. д.), для того чтобы элемент выглядел как на рис. 8.35. Поскольку стилизация очень простая, я не включаю соответствующий код в этот раздел, но если вы столкнетесь с трудностями, то всегда сможете подсмотреть решение в примере на веб-сайте по ссылке далее. Главное, о чем необходимо помнить, — что диаметр пути равен `300px`, то есть его радиус составляет `150px`.

После того как с базовыми стилями покончено, можно приступать к определению анимации. Мы хотим, чтобы аватар двигался по окружности, вдоль оранжевого пути.

Но как анимация CSS может помочь нам в создании этого эффекта? Столкнувшись с этой проблемой, многие с ходу предложат решение, подобное следующему:

```
@keyframes spin {
    to { transform: rotate(1turn); }
}

.avatar {
    animation: spin 3s infinite linear;
    transform-origin: 50% 150px; /* 150px = path radius */
}
```

Хотя это шаг в правильном направлении, аватар при этом начинает не только двигаться по окружности, но и вращаться вокруг своего центра (рис. 8.36). Например, обратите внимание, что в середине пути он оказывается вниз головой. Если бы это изображение содержало текст, то текст также переворачивался бы, и это создавало бы трудности с его прочтением. Мы хотим только, чтобы наше изображение **двигалось по окружности**, но сохраняло при этом **ориентацию относительно самого себя**.

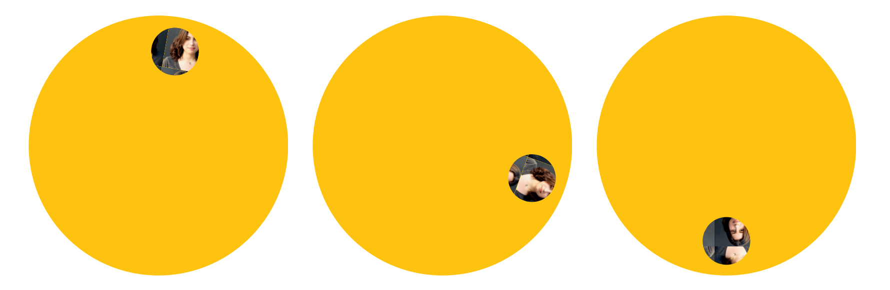


Рис. 8.36. Несколько снимков экрана нашей провальной попытки создать анимированное движение вдоль окружности

В то время ни мне, ни Крису не удалось прийти к разумному решению. Единственный способ, до которого мы сумели додуматься, заключался в том, чтобы задать множество ключевых кадров, аппроксимируя окружность, но это определенно не тянет на сколько-нибудь хорошую идею. Должен же быть лучший способ, правильно?

Решение с двумя элементами

Поварив проблему в подсознании несколько месяцев, я все же придумала решение для загадки Криса. Основная идея та же, что и в **секрете «Параллелограммы»** или в **секрете «Изображения в форме ромба»**: вложенные трансформации, отменяющие друг друга. Однако на этот раз это будет происходить не статически, а **в каждом кадре анимации**. Хитрость в том, что, как и в вышеупомянутых секретах, в данном решении нам потребуются два элемента. Следовательно, нам необходимо изменить наш исходный чистый HTML-код, добавив дополнительную обертку в форме блока `div`:

HTML

```
<div class="path">
    <div class="avatar">
        <img src="lea.jpg" />
    </div>
</div>
```

Давайте применим нашу исходную анимацию, которую мы уже тестировали выше, к обертке `.avatar`. Как видно на рис. 8.36, решение пока не работает, поскольку сам элемент также вращается. Но что, если применить к аватару **другое вращение** и **поворачивать его вокруг своей оси на тот же угол, но в противоположном направлении**? Тогда два вращения будут отменять друг друга, и мы будем видеть только движение по окружности, создаваемое разницей между центрами трансформаций!

Но мы пока не решили еще одну проблему: у нас нет статического вращения, которое мы могли бы отменить, только анимация, проходящая через целый диапазон углов. Например, если бы угол был равен `60deg`, то мы бы отменили его с помощью `-60deg` (или `300deg`), если бы это было `70deg`, то мы бы для отмены использовали `-70deg` (или `290deg`). Но если угол может быть любым в диапазоне от `0deg` до `360deg` (или от `0turn` до `1turn`, что эквивалентно), то что нам делать? Ответ намного проще, чем может казаться. Мы всего лишь определим анимацию на противоположном диапазоне (от `360deg` до `0deg`), вот так:

```
@keyframes spin {
    to { transform: rotate(1turn); }
}
@keyframes spin-reverse {
    from { transform: rotate(1turn); }
}

.avatar {
    animation: spin 3s infinite linear;
    transform-origin: 50% 150px; /* 150px = радиус пути */
}
```

```
.avatar > img {
    animation: spin-reverse 3s infinite linear;
}
```

Теперь в любой момент времени, когда первая анимация смещает аватар на `x` градусов, вторая поворачивает его на `360 - x` градусов, так как одна из них увеличивается, а вторая уменьшается. Это в точности то, чего мы стремились добиться, и, как видно на рис. 8.37, наше решение порождает желаемый эффект.

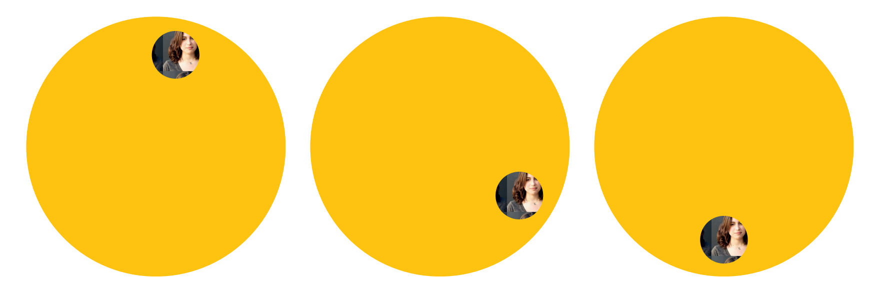


Рис. 8.37. Теперь мы достигли желаемого эффекта, но код пока что довольно неуклюжий

Код, однако, не мешало бы слегка улучшить. Мы повторяем все параметры анимации как минимум дважды. Если бы нам потребовалось изменить продолжительность анимации, нам пришлось бы отредактировать два значения, что идет вразрез с принципами DRY. Эту проблему можно легко решить, унаследовав все свойства анимации от родителя и переопределив название анимации:

```
@keyframes spin {
    to { transform: rotate(1turn); }
}
@keyframes spin-reverse {
    from { transform: rotate(1turn); }
}

.avatar {
    animation: spin 3s infinite linear;
    transform-origin: 50% 150px; /* 150px = радиус пути */
}

.avatar > img {
    animation: inherit;
    animation-name: spin-reverse;
}
```

Но почему мы используем целую новую анимацию только для того, чтобы отменить первоначальную? Помните свойство `animation-direction` из **секрета**

«Мерцание»? В том секрете мы узнали, в каких ситуациях оказывается полезным значение alternate. Здесь же мы будем использовать значение reverse, для того чтобы получить перевернутую копию нашей исходной анимации, что вообще избавит нас от необходимости создавать вторую анимацию:

```
@keyframes spin {
    to { transform: rotate(1turn); }
}

.avatar {
    animation: spin 3s infinite linear;
    transform-origin: 50% 150px; /* 150px = радиус пути */
}

.avatar > img {
    animation: inherit;
    animation-direction: reverse;
}
```

Вот и всё! Возможно, решение не идеальное, так как содержит в себе требование дополнительного элемента, но мы реализовали довольно сложную анимацию с помощью менее десятка строк CSS-кода!

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**circular-2elements**

Целиком эту дискуссию вы можете прочитать в архиве по адресу http://lists.w3.org/Archives/Public/www-style/2012Feb/0201.html.

Решение с одним элементом

Техника, описанная в предыдущем разделе, работает, но она далека от оптимальной, так как требует модификации HTML-кода. Когда я впервые пришла к этому решению, я отправила сообщение в список рассылки рабочей группы CSS (тогда я еще не была ее частью), в котором предложила добавить возможность указывать несколько центров трансформации для одного и того же элемента. Это позволило бы воплощать решения, аналогичные предыдущему, с использованием только одного элемента, да и в целом казалось весьма здравой идеей.

Дискуссия была в самом разгаре, когда Арье Грегор (Aryeh Gregor), бывший тогда одним из редакторов спецификации CSS Transforms, сделал заявление, которое поначалу сбивает с толку:

«transform-origin — это всего лишь синтаксический подсластитель. Вы всегда должны быть в состоянии вместо него использовать translate()».

— **Арье Грегор**

Но, как выяснилось, любое значение `transform-origin` можно имитировать с помощью двух трансформаций `translate()`. Например, следующие два фрагмента кода эквивалентны:

```
transform: rotate(30deg);
transform-origin: 200px 300px;
```

```
transform: translate(200px, 300px)
           rotate(30deg)
           translate(-200px, -300px);
transform-origin: 0 0;
```

```
transform: rotate(30deg);
transform-origin: 100px 50px;
```

```
transform: translate(100px,50px);
transform-origin: 0 0;
```

```
transform: translate(100px,50px)
           rotate(30deg);
transform-origin: 0 0;
```

```
transform: translate(100px,50px)
           rotate(30deg)
           translate(-100px,-50px);
transform-origin: 0 0;
```

Рис. 8.38. Как подменить центр трансформации двумя трансляциями. В каждом случае красная точка представляет центр трансформации. *Наверху*: использование transform-origin. *Внизу*: использование двух трансляций, шаг за шагом

Поначалу это кажется странным, но ситуация проясняется, если мы вспоминаем, что **функции трансформации не независимы**. Каждая из них не просто трансформирует сам элемент, к которому применяется, **она трансформирует всю систему координат этого элемента**, влияя, таким образом, на последующие трансформации. Именно поэтому мы всегда говорим, что **порядок трансформаций имеет огромное значение** и одни и те же трансформации, выполненные в разном порядке, могут приводить к разным конечным результатам. Если же вам все еще не совсем понятно, то рис. 8.38 должен помочь.

Следовательно, благодаря этой идее мы можем использовать одно и то же значение `transform-origin` для обеих наших предыдущих анимаций (нам снова придется создать отдельные анимационные эффекты, так как ключевые кадры у них теперь не совпадают):

```
@keyframes spin {
    from {
        transform: translate(50%, 150px)
                   rotate(0turn)
                   translate(-50%, -150px);
    }
    to {
        transform: translate(50%, 150px)
                   rotate(1turn)
                   translate(-50%, -150px);
    }
}
@keyframes spin-reverse {
    from {
        transform: translate(50%,50%)
                   rotate(1turn)
                   translate(-50%,-50%);
                   }
    to {
        transform: translate(50%,50%)
                   rotate(0turn)
                   translate(-50%, -50%);
    }
}

.avatar {
    animation: spin 3s infinite linear;
}

.avatar > img {
    animation: inherit;
    animation-name: spin-reverse;
}
```

Код выглядит ужасно неуклюже, но не беспокойтесь, к концу раздела мы это поправим. Обратите внимание, что теперь нам не нужны разные центры трансформаций, что было единственной причиной использования двух элементов и двух анимаций ранее. Теперь, когда у всех трансформаций общий центр, мы можем объединить две анимации в одну и работать только с `.avatar`:

```
@keyframes spin {
    from {
        transform: translate(50%, 150px)
                   rotate(0turn)
```

```
                    translate(-50%, -150px)
                    translate(50%,50%)
                    rotate(1turn)
                    translate(-50%,-50%)
    }
    to {
        transform: translate(50%, 150px)
                   rotate(1turn)
                   translate(-50%, -150px)
                   translate(50%,50%)
                   rotate(0turn)
                   translate(-50%, -50%);
    }
}
.avatar { animation: spin 3s infinite linear; }
```

Обратите внимание, что нам больше не требуются два элемента HTML: теперь мы можем просто применить класс `avatar` к самому изображению, так как мы больше не определяем для них стили по отдельности.

Определенно, код улучшается, но он все еще длинный и непонятный. Можно ли сделать его более емким? Здесь возможно несколько потенциальных усовершенствований.

Решение, лежащее на поверхности, заключается в том, чтобы объединить соседние трансформации `translate()`, в частности `translate(-50%, -150px)` и `translate(50%, 50%)`. К сожалению, процентные и абсолютные значения невозможно комбинировать (если только не прибегать к помощи функции `calc()`, но это также не улучшит читабельность кода). Однако горизонтальные трансформации отменяют друг друга, то есть, по сути, у нас здесь две трансляции по оси Y (`translateY(-150px) translateY(50%)`). Кроме того, поскольку вращения отменяют друг друга, мы можем также убрать горизонтальные трансляции до и после них и объединить вертикальные. Теперь наши ключевые кадры выглядят так:

```
@keyframes spin {
    from {
        transform: translateY(150px) translateY(-50%)
                   rotate(0turn)
                   translateY(-150px) translateY(50%)
                   rotate(1turn);
    }
    to {
        transform: translateY(150px) translateY(-50%)
                   rotate(1turn)
                   translateY(-150px) translateY(50%)
                   rotate(0turn);
    }
}
.avatar { animation: spin 3s infinite linear; }
```

Это уже короче, и повторений меньше, но все же не идеально. Можно ли сделать код еще лучше? Если в качестве начального положения аватара установить центр круга (как на рис. 8.39), то можно избавиться от первых двух трансляций, которые, по сути, помещают его в центр. Тогда анимация становится такой:

```
@keyframes spin {
    from {
        transform: rotate(0turn)
                   translateY(-150px) translateY(50%)
                   rotate(1turn);
    }
    to {
        transform: rotate(1turn)
                   translateY(-150px) translateY(50%)
                   rotate(0turn);
    }
}

.avatar { animation: spin 3s infinite linear; }
```

Кажется, это лучшее, чего мы можем достичь на сегодняшний день. Этот код нельзя назвать идеальным с точки зрения принципов DRY, но он довольно короткий. Здесь минимум повторений и нет лишних элементов HTML. Для того чтобы сделать его еще более емким и избежать повторения радиуса пути, можно воспользоваться помощью препроцессора, но я оставлю это в качестве упражнения для читателя.

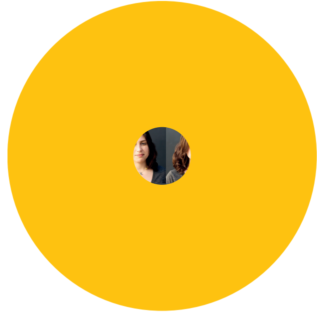


Рис. 8.39. Если в самом начале поместить аватар в центр круга, то описание ключевых кадров станет короче. Обратите внимание, что это состояние также будет служить резервным решением для случая, когда анимация не поддерживается, что может быть как желательным вариантом, так и нежелательным

ПОПРОБУЙТЕ САМИ!

http://play.csssecrets.io/**circular**

СВЯЗАННЫЕ СПЕЦИФИКАЦИИ

CSS Animations: http://w3.org/TR/css-animations

CSS Transforms: http://w3.org/TR/css-transforms

Л. Веру

Секреты CSS. Идеальные решения ежедневных задач

Перевела с английского Е. Шикарева

Заведующая редакцией	*Ю. Сергиенко*
Ведущий редактор	*Н. Римицан*
Литературный редактор	*Л. Родионова*
Корректоры	*С. Беляева, Н. Викторова*
Верстка	*Л. Егорова*

ООО «Питер Пресс», 192102, Санкт-Петербург, ул. Андреевская (д. Волкова), 3, литер А, пом. 7Н.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12.000 —
Книги печатные профессиональные, технические и научные.

Подписано в печать 24.06.16. Формат 70×100/16. Бумага писчая. Усл. п. л. 27,090. Тираж 1000. Заказ 0000.

Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор».
142300, Московская область, г. Чехов, ул. Полиграфистов, 1.
Сайт: www.chpk.ru. E-mail: marketing@chpk.ru
Факс: 8(496) 726-54-10, телефон: (495) 988-63-87